

A decorative graphic on the right side of the page. It features three concentric blue circles of different sizes. Two thin blue lines intersect at a point on the left side of the page, extending towards the top right and bottom right corners. The circles are positioned such that they appear to be part of a larger design, with the largest circle at the top right, a medium one in the middle, and a large one at the bottom right.

All-In-One Code Framework Coding Standards

by **Dan Ruder, Jialiang Ge**

This document describes the coding style guideline for native C++ and .NET (C# and VB.NET) programming used by the Microsoft All-In-One Code Framework project team.

Acknowledgement

Each chapter in this document has to acknowledge **Dan Ruder**, a Principal Escalation Engineer of Microsoft. Dan carefully reviewed every word, and contributed review comments on significant portions of the book based on his more than 20 years' programming experience. Working with the nice man has been a special treat to me.

I also thank four managers at Microsoft for continuously supporting and sponsoring the work: **Vivian Luo, Allen Ding, Felix Wu** and **Mei Liang**.

This document wouldn't contain the depth of technical details or the level of completeness it has without the input of the following people.

Alexei Levenkov, Hongye Sun, Jie Wang, Ji Zhou, Michael Sun, Kira Qian, Allen Chen, Yi-Lun Luo, Steven Cheng, Wen-Jun Zhang, Linda Liu

Some chapters derive from several Microsoft product teams' coding standards. I appreciate their sharing.

The coding standards are continuously evolving. If you discover a new best practice or a topic that is not covered, please bring that to the attention of the All-In-One Code Framework Project Group (onecode@microsoft.com). I look forward to appreciating your contributions. ☺

Disclaimer

This coding-standard document is provided "AS IS" without warranty of any kind, either expressed or implied, including but not limited to the implied warranties of merchantability and/or fitness for a particular purpose.

Please feel free to use the coding standards when you are writing VC++/VC#/VB.NET code. It would be nice, however, if you could inform us that you are using the document, or send us your feedback. You may contact us at our email address: onecode@microsoft.com.

Table of Contents

1	Overview	1
1.1	Principles & Themes	1
1.2	Terminology.....	1
2	General Coding Standards.....	3
2.1	Clarity and Consistency	3
2.2	Formatting and Style	3
2.3	Using Libraries	5
2.4	Global Variables.....	5
2.5	Variable Declarations and Initializations	5
2.6	Function Declarations and Calls	6
2.7	Statements	8
2.8	Enums	8
2.9	Whitespace	13
2.10	Braces	14
2.11	Comments	15
2.12	Regions	23
3	C++ Coding Standards	25
3.1	Compiler Options.....	25
3.2	Files and Structure.....	26
3.3	Naming Conventions	27
3.4	Pointers	30
3.5	Constants.....	31
3.6	Casting.....	32
3.7	Sizeof	32
3.8	Strings	33
3.9	Arrays.....	34
3.10	Macros.....	35
3.11	Functions	35
3.12	Structures	38
3.13	Classes	38
3.14	COM.....	44
3.15	Allocations	45
3.16	Errors and Exceptions.....	46
3.17	Resource Cleanup.....	48

3.18 Control Flow	50
4 .NET Coding Standards	54
4.1 Design Guidelines for Developing Class Libraries.....	54
4.2 Files and Structure.....	54
4.3 Assembly Properties.....	54
4.4 Naming Conventions	54
4.5 Constants.....	57
4.6 Strings	58
4.7 Arrays and Collections.....	59
4.8 Structures	61
4.9 Classes	62
4.10 Namespaces	65
4.11 Errors and Exceptions.....	65
4.12 Resource Cleanup.....	68
4.13 Interop.....	81

1 Overview

This document defines the native C++ and .NET coding standard for the [All-In-One Code Framework](#) project team. This standard derives from the experience of product development efforts and is continuously evolving. If you discover a new best practice or a topic that is not covered, please bring that to the attention of the [All-In-One Code Framework Project Group](#) and have the conclusion added to this document.

No set of guidelines will satisfy everyone. The goal of a standard is to create efficiencies across a community of developers. Applying a set of well-defined coding standards will result in code with fewer bugs, and better maintainability. Adopting an unfamiliar standard may be awkward initially, but the pain fades quickly and the benefits are quickly realized, especially when you inherit ownership of others' code.

1.1 Principles & Themes

High-quality samples exhibit the following characteristics because customers use them as examples of best practices:

1. **Understandable.** Samples must be clearly readable and straightforward. They must showcase the key things they're designed to demonstrate. The relevant parts of a sample should be easy to reuse. Samples should not contain unnecessary code. They must include appropriate documentation.
2. **Correct.** Samples must demonstrate properly how to perform the key things they are designed to teach. They must compile cleanly, run correctly as documented, and be tested.
3. **Consistent.** Samples should follow consistent coding style and layout to make the code easier to read. Likewise, samples should be consistent with each other to make them easier to use together. Consistency shows craftsmanship and attention to detail.
4. **Modern.** Samples should demonstrate current practices such as use of Unicode, error handling, defensive programming, and portability. They should use current recommendations for runtime library and API functions. They should use recommended project & build settings.
5. **Safe.** Samples must comply with legal, privacy, and policy standards. They must not demonstrate hacks or poor programming practices. They must not permanently alter machine state. All installation and execution steps must be reversible.
6. **Secure.** The samples should demonstrate how to use secure programming practices such as least privilege, secure versions of runtime library functions, and SDL-recommended project settings.

The proper use of programming practices, design, and language features determines how well samples can achieve these. This code standard is designed to help you create samples that serve as "best practices" for customers to emulate.

1.2 Terminology

Through-out this document there will be recommendations or suggestions for standards and practices. Some practices are very important and must be followed, others are guidelines that are beneficial in certain scenarios

but are not applicable everywhere. In order to clearly state the intent of the standards and practices that are discussed we will use the following terminology.

Wording	Intent	Justification
☑ Do...	This standard or practice should be followed in all cases. If you think that your specific application is exempt, it probably isn't.	These standards are present to mitigate bugs.
☒ Do Not...	This standard or practice should never be applied.	
☑ You should...	This standard or practice should be followed in most cases.	These standards are typically stylistic and attempt to promote a consistent and clear style.
☒ You should not...	This standard or practice should not be followed, unless there's reasonable justification.	
☑ You can...	This standard or practice can be followed if you want to; it's not necessarily good or bad. There are probably implications to following the practice (dependencies, or constraints) that should be considered before adopting it.	These standards are typically stylistic, but are not ubiquitously adopted.

2 General Coding Standards

These general coding standards can be applied to all languages - they provide high-level guidance to the style, formatting and structure of your source code.

2.1 Clarity and Consistency

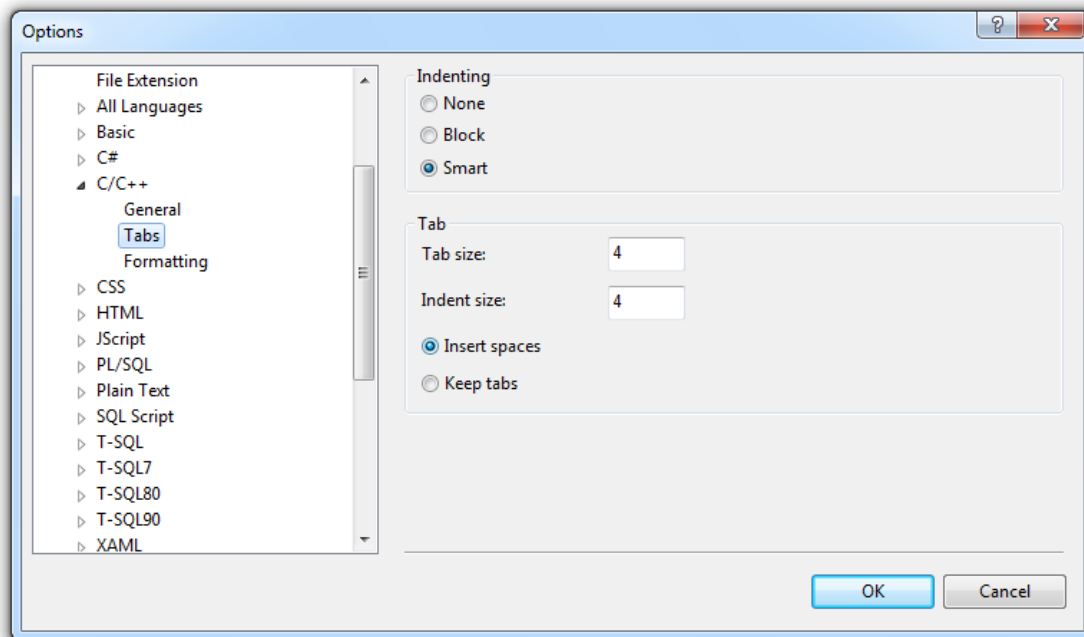
✓ **Do** ensure that clarity, readability and transparency are paramount. These coding standards strive to ensure that the resultant code is easy to understand and maintain, but nothing beats fundamentally clear, concise, self-documenting code.

✓ **Do** ensure that when applying these coding standards that they are applied consistently.

2.2 Formatting and Style

✗ **Do not** use tabs. It's generally accepted across Microsoft that tabs shouldn't be used in source files - different text editors use different spacing to render tabs, and this causes formatting confusion. All code should be written using four spaces for indentation.

Visual Studio text editor can be configured to insert spaces for tabs.



✓ **You should** limit the length of lines of code. Having overly long lines inhibits the readability of code. Break the code line when the line length is greater than column 78 for readability. If column 78 looks too narrow, use column 86 or 90.

Visual C++ sample:

```
// Get and display whether the primary access token of the process
// belongs to user account that is a member of the local Administrators
// group even if it currently is not elevated (IsUserInAdminGroup).
HWND hInAdminGroupLabel = GetDlgItem(hWnd, IDC_INADMINGROUP_STATIC);
try
{
    BOOL const fInAdminGroup = IsUserInAdminGroup();
    SetWindowText(hInAdminGroupLabel, fInAdminGroup ? L"True" : L"False");
}
catch (DWORD dwError)
{
    SetWindowText(hInAdminGroupLabel, L"N/A");
    ReportError(L"IsUserInAdminGroup", dwError);
}
```

Column 78

Visual C# sample:

```
// Get and display whether the primary access token of the process belongs
// to user account that is a member of the local Administrators group even
// if it currently is not elevated (IsUserInAdminGroup).
try
{
    bool fInAdminGroup = IsUserInAdminGroup();
    this.lbInAdminGroup.Text = fInAdminGroup.ToString();
}
catch (Exception ex)
{
    this.lbInAdminGroup.Text = "N/A";
    MessageBox.Show(ex.Message, "An error occurred in IsUserInAdminGroup",
        MessageBoxButtons.OK, MessageBoxIcon.Error);
}
```

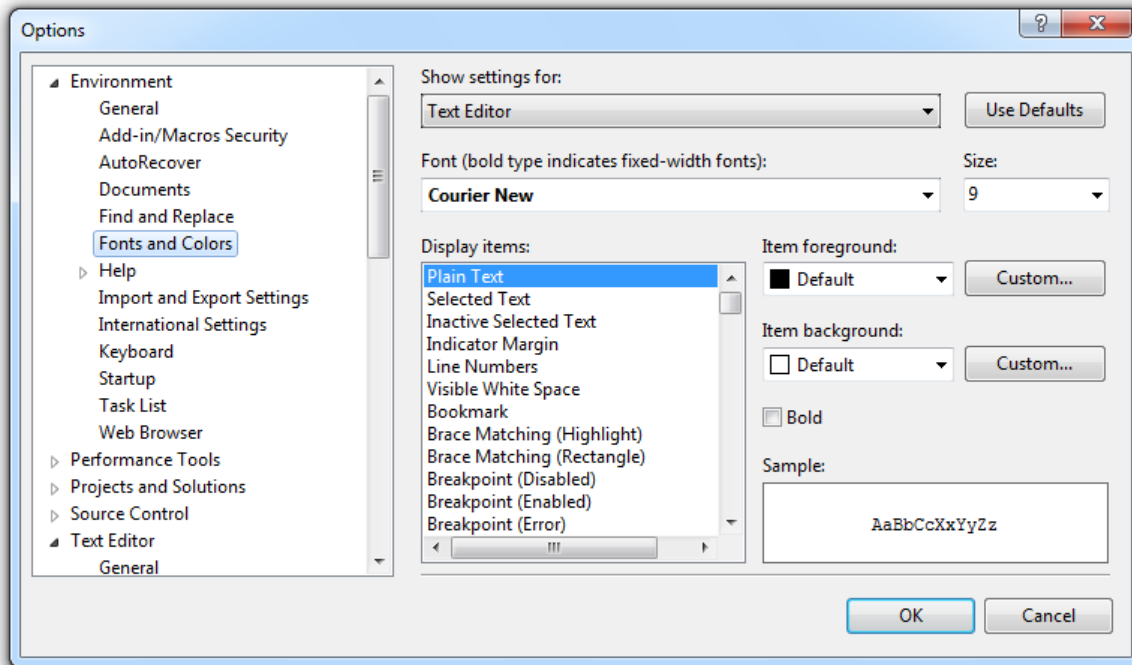
Column 86

Visual Basic sample:

```
' Get and display whether the primary access token of the process belongs
' to user account that is a member of the local Administrators group even
' if it currently is not elevated (IsUserInAdminGroup).
Try
    Dim fInAdminGroup As Boolean = Me.IsUserInAdminGroup
    Me.lbInAdminGroup.Text = fInAdminGroup.ToString
Catch ex As Exception
    Me.lbInAdminGroup.Text = "N/A"
    MessageBox.Show(ex.Message, "An error occurred in IsUserInAdminGroup",
        MessageBoxButtons.OK, MessageBoxIcon.Error)
End Try
```

Column 86

☑ **Do** use a fixed-width font, typically Courier New, in your code editor.



2.3 Using Libraries

❌ **Do not** reference unnecessary libraries, include unnecessary header files, or reference unnecessary assemblies. Paying attention to small things like this can improve build times, minimize chances for mistakes, and give readers a good impression.

2.4 Global Variables

✅ **Do** minimize global variables. To use global variables properly, always pass them to functions through parameter values. Never reference them inside of functions or classes directly because doing so creates a side effect that alters the state of the global without the caller knowing. The same goes for static variables. If you need to modify a global variable, you should do so either as an output parameter or return a copy of the global.

2.5 Variable Declarations and Initializations

✅ **Do** declare local variables in the minimum scope block that can contain them, typically just before use if the language allows; otherwise, at the top of that scope block.

✅ **Do** initialize variables when they are declared.

✅ **Do** declare and initialize/assign local variables on a single line where the language allows it. This reduces vertical space and makes sure that a variable does not exist in an un-initialized state or in a state that will immediately change.

```
// C++ sample:
HANDLE hToken = NULL;
PSID pIntegritySid = NULL;
```

```

STARTUPINFO si = { sizeof(si) };
PROCESS_INFORMATION pi = { 0 };

// C# sample:
string name = myObject.Name;
int val = time.Hours;

' VB.NET sample:
Dim name As String = myObject.Name
Dim val As Integer = time.Hours

```

❌ **Do not** declare multiple variables in a single line. One declaration per line is recommended since it encourages commenting, and could avoid confusion. As a Visual C++ example,

Good:

```

CodeExample *pFirst = NULL; // Pointer of the first element.
CodeExample *pSecond = NULL; // Pointer of the second element.

```

Bad:

```

CodeExample *pFirst, *pSecond;

```

The latter example is often mistakenly written as:

```

CodeExample *pFirst, pSecond;

```

Which is *actually* equivalent to:

```

CodeExample *pFirst;
CodeExample pSecond;

```

2.6 Function Declarations and Calls

The function/method name, return value and parameter list can take several forms. Ideally this can all fit on a single line. If there are many arguments that don't fit on a line those can be wrapped, many per line or one per line. Put the return type on the same line as the function/method name. For example,

Single Line Format:

```

// C++ function declaration sample:
HRESULT DoSomeFunctionCall(int param1, int param2, int *param3);
// C++ / C# function call sample:
hr = DoSomeFunctionCall(param1, param2, param3);
' VB.NET function call sample:
hr = DoSomeFunctionCall(param1, param2, param3)

```

Multiple Line Formats:

```

// C++ function declaration sample:

```

```

HRESULT DoSomeFunctionCall(int param1, int param2, int *param3,
    int param4, int param5);
// C++ / C# function call sample:
hr = DoSomeFunctionCall(param1, param2, param3,
    param4, param5);
' VB.NET function call sample:
hr = DoSomeFunctionCall(param1, param2, param3, _
    param4, param5)

```

When breaking up the parameter list into multiple lines, each type/parameter pair should line up under the preceding one, the first one being on a new line, indented one tab. Parameter lists for function/method *calls* should be formatted in the same manner.


```

// C++ function declaration sample:
HRESULT DoSomeFunctionCall(
    HWND hwnd,    // You can comment parameters, too
    T1 param1,    // Indicates something
    T2 param2,    // Indicates something else
    T3 param3,    // Indicates more
    T4 param4,    // Indicates even more
    T5 param5);  // You get the idea

// C++ / C# function call sample:
hr = DoSomeFunctionCall(
    hwnd,
    param1,
    param2,
    param3,
    param4,
    param5);

' VB.NET function call sample:
hr = DoSomeFunctionCall( _
    hwnd, _
    param1, _
    param2, _
    param3, _
    param4, _
    param5)

```

 **Do** order parameters, grouping the in parameters first, the out parameters last. Within the group, order the parameters based on what will help programmers supply the right values. For example, if a function takes arguments named “left” and “right”, put “left” before “right” so that their place match their names. When designing a series of functions which take the same arguments, use a consistent order across the functions. For example, if one function takes an input handle as the first parameter, all of the related functions should also take the same input handle as the first parameter.

2.7 Statements

❌ **Do not** put more than one statement on a single line because it makes stepping through the code in a debugger much more difficult.

Good:

```
// C++ / C# sample:
a = 1;
b = 2;
```

```
' VB.NET sample:
If (IsAdministrator()) Then
    Console.WriteLine("YES")
End If
```

Bad:

```
// C++ / C# sample:
a = 1; b = 2;
```

```
' VB.NET sample:
If (IsAdministrator()) Then Console.WriteLine("YES")
```

2.8 Enums

✅ **Do** use an enum to strongly type parameters, properties, and return values that represent sets of values.

✅ **Do** favor using an enum over static constants or “#define” values . An enum is a structure with a set of static constants. The reason to follow this guideline is because you will get some additional compiler and reflection support if you define an enum versus manually defining a structure with static constants.

Good:

```
// C++ sample:
enum Color
{
    Red,
    Green,
    Blue
};
```

```
// C# sample:
public enum Color
{
    Red,
    Green,
    Blue
}
```

```
' VB.NET sample:
Public Enum Color
    Red
    Green
    Blue
End Enum
```

Bad:

```
// C++ sample:
const int RED    = 0;
const int GREEN  = 1;
const int BLUE   = 2;

#define RED      0
#define GREEN    1
#define BLUE     2

// C# sample:
public static class Color
{
    public const int Red = 0;
    public const int Green = 1;
    public const int Blue = 2;
}

' VB.NET sample:
Public Class Color
    Public Const Red As Integer = 0
    Public Const Green As Integer = 1
    Public Const Blue As Integer = 2
End Class
```

❌ **Do not** use an enum for open sets (such as the operating system version, names of your friends, etc.).

✅ **Do** provide a value of zero on simple enums. Consider calling the value something like “None.” If such value is not appropriate for this particular enum, the most common default value for the enum should be assigned the underlying value of zero.

```
// C++ sample:
enum Compression
{
    None = 0,
    GZip,
    Deflate
};

// C# sample:
```

```

public enum Compression
{
    None = 0,
    GZip,
    Deflate
}

' VB.NET sample:
Public Enum Compression
    None = 0
    GZip
    Deflate
End Enum

```

❌ **Do not** use `Enum.IsDefined` for enum range checks in .NET. There are really two problems with `Enum.IsDefined`. First it loads reflection and a bunch of cold type metadata, making it a surprisingly expensive call. Second, there is a versioning issue here.

Good:

```

// C# sample:
if (c > Color.Black || c < Color.White)
{
    throw new ArgumentOutOfRangeException(...);
}

' VB.NET sample:
If (c > Color.Black Or c < Color.White) Then
    Throw New ArgumentOutOfRangeException(...)
End If

```

Bad:

```

// C# sample:
if (!Enum.IsDefined(typeof(Color), c))
{
    throw new InvalidEnumArgumentException(...);
}

' VB.NET sample:
If Not [Enum].IsDefined(GetType(Color), c) Then
    Throw New ArgumentOutOfRangeException(...)
End If

```

2.8.1 Flag Enums

Flag enums are designed to support bitwise operations on the enum values. A common example of the flags enum is a list of options.

✅ **Do** apply the `System.FlagsAttribute` to flag enums in .NET. **Do not** apply this attribute to simple enums.

✅ **Do** use powers of two for the flags enum values so they can be freely combined using the bitwise OR operation. For example,

```
// C++ sample:
enum AttributeTargets
{
    Assembly = 0x0001,
    Class    = 0x0002,
    Struct   = 0x0004
    ...
};

// C# sample:
[Flags]
public enum AttributeTargets
{
    Assembly = 0x0001,
    Class    = 0x0002,
    Struct   = 0x0004,
    ...
}

' VB.NET sample:
<Flags()> _
Public Enum AttributeTargets
    Assembly = &H1
    Class    = &H2
    Struct   = &H4
    ...
End Enum
```

✅ **You should** provide special enum values for commonly used combinations of flags. Bitwise operations are an advanced concept and should not be required for simple tasks. `FileAccess.ReadWrite` is an example of such a special value. However, **you should not** create flag enums where certain combinations of values are invalid.

```
// C++ sample:
enum FileAccess
{
    Read = 0x1,
    Write = 0x2,
    ReadWrite = Read | Write
};

// C# sample:
[Flags]
public enum FileAccess
```

```

{
    Read = 0x1,
    Write = 0x2,
    ReadWrite = Read | Write
}

' VB.NET sample:
<Flags()> _
Public Enum FileAccess
    Read = &H1
    Write = &H2
    ReadWrite = Read Or Write
End Enum

```

⊗ You should not use flag enum values of zero, unless the value represents “all flags are cleared” and is named appropriately as “None”. The following C# example shows a common implementation of a check that programmers use to determine if a flag is set (see the if-statement below). The check works as expected for all flag enum values except the value of zero, where the Boolean expression always evaluates to true.

Bad:

```

[Flags]
public enum SomeFlag
{
    ValueA = 0, // This might be confusing to users
    ValueB = 1,
    ValueC = 2,
    ValueBAndC = ValueB | ValueC,
}

SomeFlag flags = GetValue();
if ((flags & SomeFlag.ValueA) == SomeFlag.ValueA)
{
    ...
}

```

Good:

```

[Flags]
public enum BorderStyle
{
    Fixed3D          = 0x1,
    FixedSingle      = 0x2,
    None             = 0x0
}

if (foo.BorderStyle == BorderStyle.None)
{
    ...
}

```


2.9 Whitespace

2.9.1 Blank Lines

☑ **You should** use blank lines to separate groups of related statements. Omit extra blank lines that do not make the code easier to read. For example, you can have a blank line between variable declarations and code.

Good:

```
// C++ sample:
void ProcessItem(const Item& item)
{
    int counter = 0;

    if(...)
    {
    }
}
```

Bad:

```
// C++ sample:
void ProcessItem(const Item& item)
{
    int counter = 0;

    // Implementation starts here
    //
    if(...)
    {
    }

}
```

In this example of bad usage of blank lines, there are multiple blank lines between the local variable declarations, and multiple blank lines after the 'if' block.

☑ **You should** use two blank lines to separate method implementations and class declarations.

2.9.2 Spaces

Spaces improve readability by decreasing code density. Here are some guidelines for the use of space characters within code:

☑ **You should** use spaces within a line as follows.

Good:

```
// C++ / C# sample:
```

```
CreateFoo();           // No space between function name and parenthesis
Method(myChar, 0, 1); // Single space after a comma
x = array[index];      // No spaces inside brackets
while (x == y)          // Single space before flow control statements
if (x == y)             // Single space separates operators
```

```
' VB.NET sample:
CreateFoo()           ' No space between function name and parenthesis
Method(myChar, 0, 1)  ' Single space after a comma
x = array(index)      ' No spaces inside brackets
While (x = y)         ' Single space before flow control statements
If (x = y) Then       ' Single space separates operators
```

Bad:

```
// C++ / C# sample:
CreateFoo ();          // Space between function name and parenthesis
Method(myChar,0,1);    // No spaces after commas
CreateFoo( myChar, 0, 1 ); // Space before first arg, after last arg
x = array[ index ];    // Spaces inside brackets
while(x == y)          // No space before flow control statements
if (x==y)              // No space separates operators
```

```
' VB.NET sample:
CreateFoo ()           ' Space between function name and parenthesis
Method(myChar,0,1)     ' No spaces after commas
CreateFoo( myChar, 0, 1 ) ' Space before first arg, after last arg
x = array( index )     ' Spaces inside brackets
While(x = y)           ' No space before flow control statements
If (x=y) Then          ' No space separates operators
```

2.10 Braces

☑ Do use Allman bracing style in [All-In-One Code Framework](#) code samples.

The Allman style is named after Eric Allman. It is sometimes referred to as "ANSI style". The style puts the brace associated with a control statement on the next line, indented to the same level as the control statement. Statements within the braces are indented to the next level.

Good:

```
// C++ / C# sample:
if (x > 5)
{
    y = 0;
}

' VB.NET sample:
If (x > 5) Then
```

```

        y = 0
    End If

```

Bad (in All-In-One Code Framework samples):

```

// C++ / C# sample:
if (x > 5) {
    y = 0;
}

```

✅ **You should** use braces around single line conditionals. Doing this makes it easier to add code to these conditionals in the future and avoids ambiguities should the tabbing of the file become disturbed.

Good:

```

// C++ / C# sample:
if (x > 5)
{
    y = 0;
}

```

```

' VB.NET sample:
If (x > 5) Then
    y = 0
End If

```

Bad:

```

// C++ / C# sample:
if (x > 5) y = 0;

' VB.NET sample:
If (x > 5) Then y = 0

```

2.11 Comments

✅ **You should** use comments that summarize what a piece of code is designed to do and why. **Do not** use comments to repeat the code.

Good:

```

// Determine whether system is running Windows Vista or later operating
// systems (major version >= 6) because they support linked tokens, but
// previous versions (major version < 6) do not.

```

Bad:

```

// The following code sets the variable i to the starting value of the
// array. Then it loops through each item in the array.

```

✅ **You should** use `/**` comments instead of `/* */` for comments for C++ and C# code comments. The single-line syntax (`// ...`) is preferred even when a comment spans multiple lines.

```
// Determine whether system is running Windows Vista or later operating
// systems (major version >= 6) because they support linked tokens, but
// previous versions (major version < 6) do not.
if (Environment.OSVersion.Version.Major >= 6)
{
    ' Get and display the process elevation information (IsProcessElevated)
    ' and integrity level (GetProcessIntegrityLevel). The information is not
    ' available on operating systems prior to Windows Vista.
    If (Environment.OSVersion.Version.Major >= 6) Then
    End If
}
```

☑ **You should** indent comments at the same level as the code they describe.

☑ **You should** use full sentences with initial caps, a terminating period and proper punctuation and spelling in comments.

Good:

```
// Initialize the components on the Windows Form.
InitializeComponent();

' Initialize the components on the Windows Form.
InitializeComponent()
```

Bad:

```
//initialize the components on the Windows Form.
InitializeComponent();

'intialize the components on the Windows Form
InitializeComponent()
```

2.11.1 Inline Code Comments

Inline comments should be included on their own line and should be indented at the same level as the code they are commenting on, with a blank line before, but none after. Comments describing a block of code should appear on a line by themselves, indented as the code they describe, with one blank line before it and one blank line after it. For example:

```
if (MAXVAL >= exampleLength)
{
    // Reprort the error.
    ReportError(GetLastError());


    // The value is out of range, we cannot continue.
    return E_INVALIDARG;
}
```

Inline comments are permissible on the same line as the actual code only when giving a brief description of a structure member, class member variable, parameter, or a short statement. In this case it is a good idea to align the comments for all variables. For example:

```
class Example
{
public:
    ...

    void TestFunction
    {
        ...
        do
        {
            ...
        }
        while (!fFinished); // Continue if not finished.
    }

private:
    int m_length;          // The length of the example
    float m_accuracy;      // The accuracy of the example
};
```

 **You should not** drown your code in comments. Commenting every line with obvious descriptions of what the code does actually hinders readability and comprehension. Single-line comments should be used when the code is doing something that might not be immediately obvious.

The following example contains many unnecessary comments:

```
Bad:
// Loop through each item in the wrinkles array
for (int i = 0; i <= nLastWrinkle; i++)
{
    Wrinkle *pWrinkle = apWrinkles[i];    // Get the next wrinkle
    if (pWrinkle->IsNew() &&                // Process if it's a new wrinkle
        nMaxImpact < pWrinkle->GetImpact()) // And it has the biggest impact
    {
        nMaxImpact = pWrinkle->GetImpact(); // Save its impact for comparison
        pBestWrinkle = pWrinkle;           // Remember this wrinkle as well
    }
}
```

A better implementation would be:

Good:

```
// Loop through each item in the wrinkles array, find the Wrinkle with
// the largest impact that is new, and store it in 'pBestWrinkle'.
for (int i = 0; i <= nLastWrinkle; i++)
{
    Wrinkle *pWrinkle = apWrinkles[i];
    if (pWrinkle->IsNew() && nMaxImpact < pWrinkle->GetImpact())
    {
        nMaxImpact = pWrinkle->GetImpact();
        pBestWrinkle = pWrinkle;
    }
}
```

☑ **You should** add comments to call out non-intuitive or behavior that is not obvious from reading the code.

2.11.2 File Header Comments

☑ **Do** have a file header comment at the start of every human-created code file. The header comment templates are as follows:

VC++ and VC# file header comment template:

```
/****** Module Header *****/
Module Name:  <File Name>
Project:      <Sample Name>
Copyright (c) Microsoft Corporation.

<Description of the file>

This source is subject to the Microsoft Public License.
See http://www.microsoft.com/opensource/licenses.mspx#Ms-PL.
All other rights reserved.

THIS CODE AND INFORMATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND,
EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR PURPOSE.
/*****/
```

VB.NET file header comment template:

```
'***** Module Header *****/
' Module Name:  <File Name>
' Project:      <Sample Name>
' Copyright (c) Microsoft Corporation.
'
' <Description of the file>
'
' This source is subject to the Microsoft Public License.
' See http://www.microsoft.com/opensource/licenses.mspx#Ms-PL.
```

```
' All other rights reserved.
'
' THIS CODE AND INFORMATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND,
' EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED
' WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR PURPOSE.
' *****/
```

For example,

```
/****** Module Header *****/
Module Name: CppUACSelfElevation.cpp
Project: CppUACSelfElevation
Copyright (c) Microsoft Corporation.
```

User Account Control (UAC) is a new security component in Windows Vista and newer operating systems. With UAC fully enabled, interactive administrators normally run with least user privileges. This example demonstrates how to check the privilege level of the current process, and how to self-elevate the process by giving explicit consent with the Consent UI.

This source is subject to the Microsoft Public License.
See <http://www.microsoft.com/opensource/licenses.mspx#Ms-PL>.
All other rights reserved.

```
THIS CODE AND INFORMATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND,
EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR PURPOSE.
\*****/
```

2.11.3 Class Comments

☒ **You should** provide banner comments for all classes and structures that are non-trivial. The level of commenting should be appropriate based on the audience of the code.

C++ class comment template:

```
//
// NAME: class <Class name>
// DESCRIPTION: <Class description>
//
```

C# and VB.NET use .NET descriptive XML Documentation comments. When you compile .NET projects with /doc the compiler will search for all XML tags in the source code and create an XML documentation file.

C# class comment template:

```
/// <summary>
/// <Class description>
```

```
/// </summary>
```

VB.NET class comment template:

```
''' <summary>
''' <Class description>
''' </summary>
```

For example,

```
//
//  NAME: class CodeExample
//  DESCRIPTION: The CodeExample class represents an example of code, and
//  tracks the length and complexity of the example.
//
class CodeExample
{
    ...
};

/// <summary>
/// The CodeExample class represents an example of code, and tracks
/// the length and complexity of the example.
/// </summary>
public class CodeExample
{
    ...
}
```

2.11.4 Function Comments

☑ **You should** provide banner comments for all public and non-public functions that are not trivial. The level of commenting should be appropriate based on the audience of the code.

C++ function comment template:

```
/*-----
FUNCTION: <Function prototype>

PURPOSE:
    <Function description>

PARAMETERS:
    <Parameter name> -<Parameter description>

RETURN VALUE:
    <Description of function return value>
```



```

EXCEPTION:
    <Exception that may be thrown by the function>

EXAMPLE CALL:
    <Example call of the function>

REMARKS:
    <Additional remarks of the function>
-----*/

```

C# and VB.NET use descriptive XML Documentation comments. At least a <summary> element and also a <parameters> element and <returns> element, where applicable, are required. Methods that throw exceptions should make use of the <exception> element to indicate to consumers what exceptions may be thrown.

C# function comment template:

```

/// <summary>
/// <Function description>
/// </summary>
/// <param name="Parameter name">
/// <Parameter description>
/// </param>
/// <returns>
/// <Description of function return value>
/// </returns>
/// <exception cref="<Exception type>">
/// <Exception that may be thrown by the function>
/// </exception>

```

VB.NET function comment template:

```

''' <summary>
''' <Function description>
''' </summary>
''' <param name="Parameter name">
''' <Parameter description>
''' </param>
''' <returns>
''' <Description of function return value>
''' </returns>
''' <exception cref="<Exception type>">
''' <Exception that may be thrown by the function>
''' </exception>

```

For example,

```

/*-----

```

FUNCTION: IsUserInAdminGroup(HANDLE hToken)

PURPOSE:

The function checks whether the primary access token of the process belongs to user account that is a member of the local Administrators group, even if it currently is not elevated.

PARAMETERS:

hToken - the handle to an access token.

RETURN VALUE:

Returns TRUE if the primary access token of the process belongs to user account that is a member of the local Administrators group. Returns FALSE if the token does not.

EXCEPTION:

If this function fails, it throws a C++ DWORD exception which contains the Win32 error code of the failure.

EXAMPLE CALL:

```
try
{
    if (IsUserInAdminGroup(hToken))
        wprintf (L"User is a member of the Administrators group\n");
    else
        wprintf (L"User is not a member of the Administrators group\n");
}
catch (DWORD dwError)
{
    wprintf(L"IsUserInAdminGroup failed w/err %lu\n", dwError);
}

-----*/

/// <summary>
/// The function checks whether the primary access token of the process
/// belongs to user account that is a member of the local Administrators
/// group, even if it currently is not elevated.
/// </summary>
/// <param name="token">The handle to an access token</param>
/// <returns>
/// Returns true if the primary access token of the process belongs to
/// user account that is a member of the local Administrators group.
/// Returns false if the token does not.
/// </returns>
/// <exception cref="System.ComponentModel.Win32Exception">
/// When any native Windows API call fails, the function throws a
/// Win32Exception with the last error code.
```

```
/// </exception>
```

Any method or function which can fail with side-effects should have those side-effects clearly communicated in the function comment. As a general rule, code should be written so that it has no side-effects in error or failure cases; the presence of such side-effects should have some clear justification when the code is written. (Such justification is not necessary for routines which zero-out or otherwise overwrite some output-only parameter.)

2.11.5 Commenting Out Code

Commenting out code is necessary when you demonstrate multiple ways of doing something. The ways except the first one are commented out. Use [-or-] to separate the multiple ways. For example,

```
// C++ / C# sample:
// Demo the first solution.
DemoSolution1();

// [-or-]

// Demo the second solution.
//DemoSolution2();

' VB.NET sample:
' Demo the first solution.
DemoSolution1();

' [-or-]

' Demo the second solution.
'DemoSolution2();
```

2.11.6 TODO Comments

☒ **Do not** use TODO comments in any released samples. Every sample must be complete and not require a list of unfinished tasks sprinkled throughout the code.

2.12 Regions

☒ **Do** use region declarations where there is a large amount of code that would benefit from this organization. Grouping the large amount of code by scope or functionality improves readability and structure of the code.

C++ regions:

```
#pragma region "Helper Functions for XX"
...
#pragma endregion
```

C# regions:

```
#region Helper Functions for XX
```

```
...  
#endregion
```

VB.NET regions:

```
#Region "Helper Functions for XX"  
...  
#End Region
```

3 C++ Coding Standards

These coding standards can be applied to native C++.

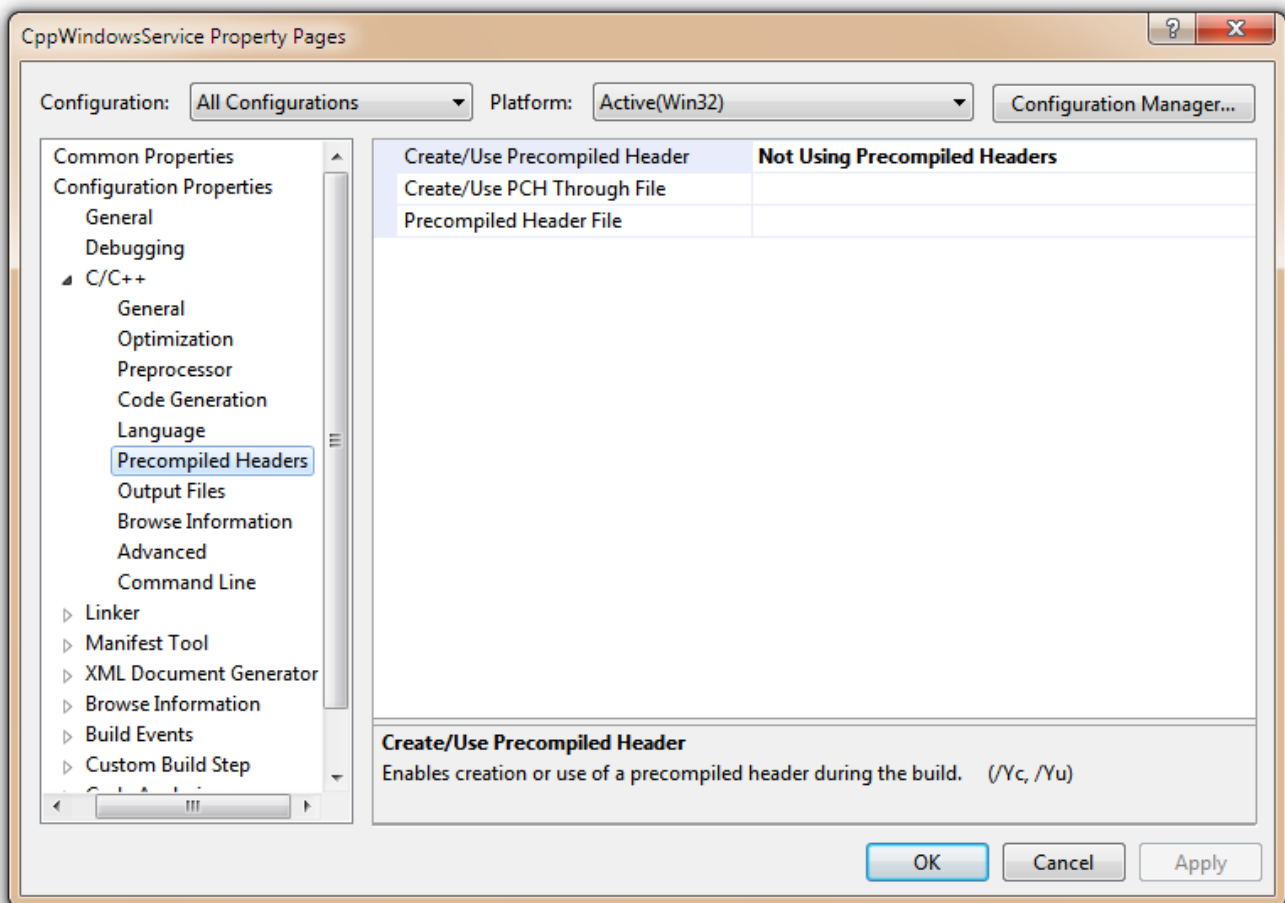
3.1 Compiler Options

3.1.1 Precompiled Header

☒ **Do not** use precompiled headers.

By default, Visual C++ projects use precompiled headers. This is a system whereby the large Windows headers are compiled only once when you build `stdafx.h/cpp`. Every other `.CPP` file in your project needs to `#include "stdafx.h"` as the first `#include` in order to build. The compiler specifically looks for the name `"stdafx.h"` to know when to insert the precompiled header information.

In code samples, precompiled header must be turned off. In your project options, go to the C/C++ tab and select the Precompiled headers category. Click the Not using precompiled headers radio button, and then click OK. Make sure to modify All Configurations (including both Debug and Release configurations). Then, remove `#include<stdafx.h>` from all source files.



3.1.2 Enable All Warnings, and Treat Them as Errors

✓ **You should** compile all code at the highest warning level.

✓ **You should** treat all warnings as errors.

The warnings provided by the compiler are often useful in identifying bad practices, or even subtle bugs. You can use the compiler warnings as an extra level of verification on your code.

In Visual Studio you can enable warning level four in the properties for your project; on the 'Property Pages' for your project, go to "Configuration Properties", "C/C++", "General" and set "Warning Level" to "Level 4".

3.2 Files and Structure

3.2.1 stdafx.h, stdafx.cpp, targetver.h

✓ **You should** delete the stdafx.h, stdafx.cpp and targetver.h files generated by Visual Studio project template to keep the code files simple. However, if you have a lot of standard header files to be shared by many code files, you may create a single header file to include them, much like windows.h.

3.2.2 Header Files

✓ **Do** use include guards within a header file (internal include guards) to prevent unintended multiple inclusions of the header file.

The `#ifndef` and `#endif` from the example, below, should be the first and last lines of the header file. The following example shows how to use "`#ifndef/#endif`" as an include guard in "CodeExample.h";

```
// File header comment goes first ...

#ifndef CODE_EXAMPLE_H_
#define CODE_EXAMPLE_H_

class CodeExample
{
    ...
};

#endif
```

You can also use "`#pragma once`" (a Microsoft Compiler specific extension) as an alternative to "`#ifndef/#endif`" include guard:

```
// File header comment goes first ...

#pragma once

class CodeExample
{
```

```
...
};
```

❌ **You should not** implement functions in header files. Header files should only contain the declarations of functions and data structures. Their implementation should be in the .cpp files.

3.2.3 Implementation Files

Implementation files contain the actual function bodies for global functions, local functions, and class method functions. An implementation file has the extension .c or .cpp. Note that an implementation file does not have to implement an entire module. It can be split up and #include a common internal interface.

✅ **You should** keep declarations that don't have to be exported inside the implementation file. Furthermore, you should add the static keyword to limit their scope to just the compilation unit defined by the .cpp/.c file. This will reduce changes of getting “multiply-defined symbol” errors during linking when two or more .cpp files use the same internal variables.

3.3 Naming Conventions

3.3.1 General Naming Conventions

✅ **Do** use meaningful names for various types, functions, variables, constructs and data structures. Their use should be plainly discernable from their name alone.

Single-character variables should only be used as counters (i, j) or as coordinates (x, y, z). As a rule-of-thumb a variable should have a more descriptive name as its scope increases.

❌ **You should not** use shortenings or contractions as parts of identifier names. For example, use “GetWindow” rather than “GetWin”. For functions of common types, thread procs, window procedures, dialog procedures use the common suffixes for these “ThreadProc”, “DialogProc”, “WndProc”.

3.3.2 Capitalization Naming Rules for Identifiers

The following table describes the capitalization and naming rules for different types of identifiers.

Identifier	Casing	Naming Structure	Example
Class	PascalCasing	Noun	<code>class</code> ComplexNumber {...}; <code>class</code> CodeExample {...}; <code>class</code> StringList {...};
Enumeration	PascalCasing	Noun	<code>enum</code> Type {...};
Function, Method	PascalCasing	Verb or Verb-Noun	<code>void</code> Print() <code>void</code> ProcessItem()
Interface	PascalCasing 'I' prefix	Noun	<code>class</code> IDictionary {...};
Structure	All capital, separate words with '_'	Noun	<code>struct</code> FORM_STREAM_HEADER

Macro, Constant	All capital, separate words with '_'		<code>#define BEGIN_MACRO_TABLE(name) ...</code> <code>#define MACRO_TABLE_ENTRY(a, b, c) ...</code> <code>#define END_MACRO_TABLE() ...</code> <code>const int BLACK = 3;</code>
Parameter, Variable	camelCasing	Noun	<code>exampleText, dwCount</code>
Template parameter	PascalCasing 'T' prefix	Noun	<code>T, TItem, TPolicy</code>

3.3.3 Hungarian Notation

✓ **You can** use Hungarian notation in parameter and variable names. However, Hungarian notation is a relic that makes code refactoring harder; i.e. change a type of a variable and you need to rename it everywhere.

The following table defines a set of suitable Hungarian notation tags should you choose to use Hungarian notation.

Type	Tag	Description
bool, BOOL, bitfield	f	A flag. For example, BOOL fSucceeded;
BYTE		An 8 bit unsigned quantity. The use of BYTES should be limited to opaque quantities, like cookies, bitfields, etc.
WORD		An unsigned 16 bit quantity. The use of WORDs should be limited to opaque quantities, like cookies, handles, bitfields, etc.
DWORD	dw	An unsigned 32 bit quantity. The use of DWORDs should be limited to opaque quantities, like cookies, handles, bitfields, etc.
HRESULT	hr	HRESULT values are commonly used through-out Win32 for error or status values.
VARIANT	vt	An OLE VARIANT.
HANDLE	h	A handle.
int, unsigned int		A 32 bit ordinal number (can be compared using <, <=, >, >=). NOTE: on 64 bit versions of Windows integer is 32 bits.
short, unsigned short		A 16-bit ordinal number. These tags should be rarely used; they are acceptable in structures for disk formats and the heap.
long, unsigned long		A 32-bit ordinal number. These tags should be rarely used, as "int" accomplishes the same thing and is preferred to "long".
__int64, LONGLONG, ULONGLONG		A 64-bit ordinal number.
TCHAR, wchar_t, char	ch	A character (sign unspecified). The "wchar_t" type is the preferred for wide characters as it's a C++ construct. We do not have different tags for char's and TCHARs's because we use Unicode through the project. In the rare case of a function that contains both char's and WCHAR's, use "ch" for char and "wch" for wchar_t.
PWSTR, PCWSTR,	psz	A pointer to a zero-terminated string. Since we are using Unicode

wchar_t *, PSTR, PCSTR, char *		throughout the project, we do not have different tags for PSTRs and PWSTR's. We do not have different tags for char's and TCHARs's because we use Unicode through the project. In the rare case of a function that contains both PSTR's and PWSTRs, use "psz" for PSTR and "pwsz" for PWSTR. Outside of MIDL use PWSTR and PSTR, even for interface methods; all pointers are long and the L prefix is obsolete.
wchar_t [], char []	sz	A zero-terminated string in the form of a character array on the stack. For example, wchar_t szMessage[BUFFER_SIZE];
BSTR	bstr	An OLE Automation BSTR.
void		A void. Use the "p" prefix for a pointer to void.
(*)()		A function. Use the "p" prefix for a pointer to function.

For example,

```
HANDLE hMapFile = NULL;
DWORD dwError = NO_ERROR;
```

Hungarian Prefixes can be used to augment the type information – the prefix is used with the Hungarian tag.

Prefix	Description
p	A pointer (32bit or 64 bit depending on platform).
sp	A 'smart' pointer, i.e. a class that has pointer-like semantics.
c	A count. For example, cbBuffer means the byte count of a buffer. It is acceptable if "c" is not followed by a tag.
m_	A member variable in a class.
s_	A static member variable in a class.
g_	A global variable.
I	COM interface

For example,

```
UINT cch; // Count of characters
PWSTR psz; // String pointer, null terminated
wchar_t szString[] = L"foo";
```

3.3.4 UI Control Naming Conventions

UI controls may use the following prefixes and follow the resource ID formats. The primary purpose is to make code more readable.

Control Type	Control Handle Name Format	MFC Control Name Format	Resource ID (All capital, separate words with '_')
Animation Control	hXxxAnimate	aniXxx	IDC_ANIMATE_XXX

Button	hXxxButton	btnXxx	IDC_BUTTON_XXX
Check Box	hXxxCheck	chkXxx	IDC_CHECK_XXX
ComboBox	hXxxCombo	cmbXxx	IDC_COMBO_XXX
Date Time Picker	hXxxDatePicker	dtpXxx	IDC_DATETIMEPICKER_XXX
Edit Control	hXxxEdit	tbXxx	IDC_EDIT_XXX
Group Box	hXxxGroup	grpXxx	IDC_STATIC_XXX
Horizontal Scroll Bar	hXxxScroll	hsbXxx	IDC_SCROLLBAR_XXX
IP Address Control	hXxxIpAddr	ipXxx	IDC_IPADDRESS_XXX
List Box	hXxxList	lstXxx	IDC_LIST_XXX
List(View) Control	hXxxList	lvwXxx	IDC_LIST_XXX
Menu	hXxxMenu	N/A	IDM_XXX
Month Calendar Control	hXxxCalendar	mclXxx	IDC_MONTHCALENDAR_XXX
Picture Box	hXxxPicture	pctXxx	IDC_STATIC_XXX
Progress Control	hXxxProgress	prgXxx	IDC_PROGRESS_XXX
Radio Box	hXxxRadio	radXxx	IDC_RADIO_XXX
Rich Edit Control	hXxxRichEdit	rtfXxx	IDC_RICHEDIT_XXX
Slider Control	hXxxSlider	sldXxx	IDC_SLIDER_XXX
Spin Control	hXxxSpin	spnXxx	IDC_SPIN_XXX
Static Text	hXxxLabel	lbXxx	IDC_STATIC_XXX
SysLink Control	hXxxLink	lnkXxx	IDC_SYSLINK_XXX
Tab Control	hXxxTab	tabXxx	IDC_TAB_XXX
Tree(View) Control	hXxxTree	tvwXxx	IDC_TREE_XXX
Vertical Scroll Bar	hXxxScroll	vsbXxx	IDC_SCROLLBAR_XXX

3.4 Pointers

✓ **You should** always initialize pointers when you declare them and you should reinitialize them to NULL or other invalid value after freeing them. This prevents the rest of the code from using an uninitialized pointer to corrupt the process's address space by accidentally reading/writing to an unknown location. For example:

Good:

```
BINARY_TREE *directoryTree = NULL;
DWORD *pdw = (DWORD *)LocalAlloc(LPTR, 512);
...
if (pdw != NULL)
{
    LocalFree(pdw);
    pdw = NULL;
}
...
```

```

if (directoryTree != NULL)
{
    // Free directoryTree with match to way it was allocated
    FreeBinaryTree(directoryTree);
    directoryTree = NULL;
}

```

☑ **You should** put a space between the '*' character(s) and the type when specifying a pointer type/variable, but there should be no space between the '*' character(s) and the variable. Setting this rule is to be consistent and uniform in code. Here are some examples:

Good:

```

HRESULT GetInterface(IStdInterface **ppSI);
INFO *GetInfo(DWORD *pdwCount);
DWORD *pdw = (DWORD *)pv;
IUnknown *pUkwn = static_cast<IUnknown *>(*ppv);

```

Bad:

```

HRESULT GetInterface(IStdInterface** ppSI);
INFO* GetInfo(DWORD * pdwCount);
DWORD* pdw = (DWORD*)pv;
IUnknown* pUkwn = static_cast<IUnknown*>(*ppv);

```

3.5 Constants

☑ **Do** define named constants as 'const' values, instead of "#define" values. For example:

Good:

```
const int BLACK = 3;
```

Bad:

```
#define BLACK 3
```

When you use const values, the compiler will enforce type checking and add the constants to the symbol table, which makes debugging easier. In contrast, the preprocessor does neither.

☑ **You should** define groups of related constants using enum. This allows the group of constants to share a unique type and improves function interfaces. For example:

Good:

```
enum DayOfWeek {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
enum Color {Black, Blue, White, Red, Purple};
```

```

// Note the strong type parameter checking; calling code can't reverse them.
BOOL ColorizeCalendar (DayOfWeek today, Color todaysColor);

```

Bad:

```
const int Sunday = 0;
const int Monday = 1;
const int Black = 0;
const int Blue = 1;

// Note the weak type parameter checking; calling code can reverse them.
BOOL ColorizeCalendar (int today, int todaysColor);
```

✅ **You should** use ‘const’ when passing and returning parameters where appropriate. By applying ‘const’ the intent of the code is clearly spelled out, and the compiler can provide an added level of verification that the code isn’t modifying values that it shouldn’t be.

Const usage	Meaning	Description
const int *x;	Pointer to a const int	Value pointed to by x can’t change
int * const x;	Const pointer to an int	x cannot point to a different location.
const int *const x;	Const pointer to a const int	Both the pointer and the value pointed to cannot change.

3.6 Casting

✅ **You should** use C++ casts. The C++ casts are more explicit, give finer-grain control and express the intent of the code more clearly. There are three types of C++ casts:

1. `static_cast` handles related types such as one pointer to another in the same hierarchy. This is a safe cast. The compiler will ensure that the type is actually what you are casting to. This is often needed to disambiguate the type of an item when it is multiply derived.
2. `reinterpret_cast` handles conversion between unrelated types. Warning: Do not cast from a `DWORD` to a pointer or visa-versa. It will not compile under 64 bits. Do comment your `reinterpret_cast<>` usage; this is needed to relieve the concern that future readers will have when they see the cast.
3. `const_cast` is used to cast away the ‘const’ness of an object.

The syntax for all three is similar:

```
DerivedClass *pDerived = HelperFunction();
BaseClass *pBase = static_cast<BaseClass *>(pDerived);
```

❌ **You should not** use ‘const_cast’ unless absolutely necessary. Having to use ‘const_cast’ typically means that an API is not using ‘const’ appropriately. Note; The Win32 API doesn’t always use ‘const’ for passing parameters and it may be necessary to use `const_cast` when using the Win32 API.

3.7 Sizeof

✅ **Do** use `sizeof(var)` instead of `sizeof(TYPE)` whenever possible. To be explicit about the size value being used whenever possible write `sizeof(var)` instead of `sizeof(TYPE_OF_VAR)`. Do not code the known size or type of the variable. Do reference the variable name instead of the variable type in `sizeof`:

Good:

```
MY_STRUCT s;
ZeroMemory(&s, sizeof(s));
```

Bad:

```
MY_STRUCT s;
ZeroMemory(&s, sizeof(MY_STRUCT));
```

❌ **Do not** use sizeof for arrays to get the element number. Use ARRAYSIZE.

3.8 Strings

✅ **Do** write explicitly UNICODE code because it enables easier globalization. Don't use TCHAR or ANSI code because it eliminates half of all testing and eliminates string bugs. This means:

Use the wide char types including wchar_t, PWSTR, PCWSTR instead of the TCHAR versions.

Good:

```
HRESULT Function(PCWSTR)
```

Bad:

```
HRESULT Function(PCTSTR)
```

Variable names should not indicate "W" in the name.

Good:

```
Function(PCWSTR psz)
```

Bad:

```
Function(PCWSTR pwsz)
```

Don't use the TEXT macro, instead use the L prefix for creating Unicode string constants L"string value".

Good:

```
L"foo"
```

Bad:

```
TEXT("foo")
```

Prefer wchar_t to WCHAR because it is the native C++ type.

Good:

```
L"foo"
wchar_t szMessage[260];
```

Bad:

```
TEXT("foo")
WCHAR szMessage[260];
```

Never explicitly use the A/W versions of the APIs. This is bad style because the names of the APIs actually are the base name without A/W. And, it makes code hard to port when you do need to switch between ANSI/Unicode.

Good:

```
CreateWindow(...);
```

Bad:

```
CreateWindowW(...);
```

✅ **You should** use fixed size stack buffers for string instead of allocating when possible. There are several benefits to using a fixed sized stack buffer rather than allocating a buffer:

- Fewer error states, no need to test for allocation failure and write code to handle this.
- No opportunity to leak, the stack is trivially cleaned up for you.
- Better performance, no transient heap usage.

There are cases where a stack buffer should be avoided

- When the data size can be arbitrary and thus in some cases it will not fit.
Note that UI strings are limited by UA guidelines and the size of the screen, so you can usually pick a fixed upper bound for the size. It is best to double the size you think the string will be to accommodate future edits and growth in other languages (the rule there is 30% for language growth).
- “Large data”; rule of thumb is several times larger than MAX_PATH (260) or more than one MAX_URL (2048) should not be on the stack.
- Recursive functions.

So for small size data where it is reasonable to pick a max size it is best to put this data on the stack.

3.9 Arrays

3.9.1 Array Size

✅ **Do** use ARRAYSIZE() as the preferred way to get the size of an array. ARRAYSIZE() is declared in a way that produces an error if it is used on a non-array type, resulting in error C2784. For anonymous types you need to use the less safe _ARRAYSIZE() macro. ARRAYSIZE() should be used instead of RTL_NUMBER_OF(), _countof(), NUMBER_OF(), etc.

✅ **Do** derive the array size from the variable rather than specifying the size in your code.

Good:

```
ITEM rgItems[MAX_ITEMS];
for (int i = 0; i < ARRAYSIZE(rgItems); i++) // use ARRAYSIZE()
{
    rgItems[i] = fn(x);
    cb = sizeof(rgItems[i]); // specify the var, not its type
}
```

Bad:

```
ITEM rgItems[MAX_ITEMS];
// WRONG, use ARRAYSIZE(), no need for MAX_ITEMS typically
for (int i = 0; i < MAX_ITEMS; i++)
{
    rgItems[i] = fn(x);
    cb = sizeof(ITEM); // WRONG, use var instead of its type
}
```

3.9.2 Array Initialization

✅ **Do** use "`= {}`" to zero array memory. The compiler optimizer does better with "`= {}`" than "`= {0}`" and `ZeroMemory`, so "`= {}`" is preferred.

3.10 Macros

❌ **You should not** use macros unless they are absolutely necessary. Most functionality that is typically achieved through macros can be implemented other C++ constructs (using constants, enums, inline functions, or templates) which will yield clearer, safer, and more understandable code.

Good:

```
__inline PCWSTR PCWSTRFromBSTR(__in BSTR bstr)
{
    return bstr ? bstr : L"";
}
```

Bad:

```
#define PCWSTRFromBSTR(bstr) (bstr ? bstr : L"")
```

❌ **Do not** use "`#define`" values for constant values. See the section on "Constants and 'const'".

❌ **Do not** use the following existing macros: `SIZEOF()`, `IID_PPV_ARG()` (use `IID_PPV_ARGS()` instead).

3.11 Functions

3.11.1 Validating Parameters

✅ **Do** validate parameters to functions that will be used by the public. If the parameters are not valid, set the last error `ERROR_INVALID_PARAMETER` or return the `HRESULT E_INVALIDARG`.

3.11.2 Reference Parameters

❌ **Do not** use ref parameters for output because it makes it hard to determine whether a variable is modified (an output) at the call site. Use pointers instead. For example, consider:

```
Function()
{
    int n = 3;
```

```

    Mystery(n);
    ASSERT(n == 3);
}

```

Is the assertion valid? If you missed the declaration of the function `Mystery`, you might think the answer is, "Of course it is. The value of `n` is never modified". However, if the function `Mystery` were declared as:

```
void Mystery(int &n);
```

Then the answer is, "Maybe, maybe not". If the `Mystery` function intends to modify its argument, it should be rewritten as:

```

void Mystery(int *pn);

Function()
{
    int n = 3;
    Mystery(&n);
}

```

It is now clearer that the `Mystery` function can change its argument.

If you choose to pass an object by reference (for example, because it is a structure), either pass it explicitly by pointer (if it is an output parameter), or use a const reference (if it is an input parameter). The const attribute indicates that the object is not modified by the function. This preserves the rule that "a parameter passed without a `&` is not modified by the function." We have defined macros for common cases of object types, like `REFCLSID`, `REFIID` and `REFPROPERTYKEY`.

3.11.3 Unreferenced Parameters

When implementing methods in an interface or standard export it is common for some of the parameters to not be referenced. The compiler detects unused parameters and will produce a warning that some components treat as an error. To avoid this comment out the unused parameter using the `/* param_name */` syntax, don't use the `UNREFERENCED_PARAMETER()` macro since that is 1) less concise, 2) does not ensure that the parameter is in fact unreferenced.

Good:

```
LRESULT WndProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM /* lParam */)
```

Bad:

```

LRESULT WndProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    UNREFERENCED_PARAMETER(lParam);
    ...
}

```


3.11.4 Output String Parameters

A common way to return a string from a function is to have the caller specify the address where the value should be stored and the length of that buffer as a count of characters. This is the “pszBuf/cchBuf” pattern. In the method, you need to explicitly test the buffer size > 0 first.

In COM applications, you can return __out string parameters as strings allocated by CoTaskMemAlloc / SysAllocString. This avoids string size limitations in the code. The caller is responsible for calling CoTaskMemFree / SysFreeString.

3.11.5 Return Values

✓ **Do** test the return of a function, not the out parameters, in the caller. Some functions communicate the success or failed state in multiple ways; for example in COM methods it is visible in the HRESULT and the out parameter. For example both of the following are correct.

Good:

```
IShellItemImageFactory *psiif;
if (SUCCEEDED(psi->QueryInterface(IID_PPV_ARGS(&psiif))))
{
    // Use psiif
    psiif->Release();
}
```

Bad:

```
IShellItemImageFactory *psiif;
psi->QueryInterface(IID_PPV_ARGS(&psiif));
if (psiif)
{
    // Use psiif
    psiif->Release();
}
```

The reasons are:

- The HRESULT is the more prominent result from the function and is the most appropriate to test.
- Usually the value of the HRESULT carries important information that needs to be propagated, this reduces the chance of mapping or treating all failures to E_FAIL.
- Testing the HRESULT makes this code consistent with cases where the out params are not zeroed on failure (many win32 APIs).
- Testing the HRESULT enables putting the call and the test on the same line.
- Testing the HRESULT is more efficient in terms of code generation.

3.12 Structures

3.12.1 Typedef Structures

✅ **You should** use typedef when you need to create a named type. If all you want is a single struct variable, you don't need the typedef. The typedef tag should be the same name as the type, "typedef struct FOO { int x; } FOO;" if it is needed (forward use and self referencing typedefs need the struct tag). Structure names are all upper case. Separate words with '_'. For example,

```
// The format of the bytes in the data stream....
typedef struct FORM_STREAM_HEADER
{
    int cbTotalSize;
    int cbVarDataOffset;
    USHORT usVersion;
} FORM_STREAM_HEADER;
```

3.12.2 Structure Initialization

✅ **Do** use "= {}" to zero structure memory.

```
PROPVARIANT pv = {};
```

When a structure has a byte size field as the first member, you could use the following shortcut to initialize the size field and zero initialize the other fields:

```
SHELLEXECUTEINFO sei = { sizeof(sei) };
sei.lpFile = ...
```

3.12.3 Structures vs. Classes

✅ **Do** use a structure to define a data aggregate that does not contain functions. Use a class if the data structure includes member functions. In C++, a struct can have member functions and operators and everything else that a class can have. In fact, the only difference between a class and a struct is that all members default to public access in a struct but private access in a class. To match the normal intuition, we use a class if and only if there are member functions included.

3.13 Classes

3.13.1 Data Members

❌ **Do not** declare public data members. Use inline accessor functions for performance.

✅ **Do** prefer initialization to assignment in constructors. For example, using initialization:

```
Good:
class Example
{
```

```

public:
    Example(const int length, const wchar_t *description) :
        m_length(length),
        m_description(description),
        m_accuracy(0.0)
    {
    }

private:
    int m_length;
    CString m_description;
    float m_accuracy;
};

```

Bad:

```

class Example
{
public:
    Example(int length, const wchar_t *description)
    {
        m_length = length;
        m_description = description;
        m_accuracy = 0.0;
    }

private:
    int m_length;
    CString m_description;
    float m_accuracy;
};

```

✅ **Do** initialize member variables in the same order that they were defined in the class declaration. The order of initialization is the order the members are declared in the class definition, not the order of the initialization list. If both orders are consistent then the code will reflect what will be generated by the compiler. As an example, consider the “CodeExample” class.

Bad:

```

class CodeExample
{
public:
    explicit CodeExample(int size);
    ~CodeExample();

private:
    wchar_t *m_buffer;
    int m_size;
};

```

```

CodeExample::CodeExample(int size) :
    m_size(size),
    m_buffer((wchar_t*)operator new[] (m_size))
{
}

CodeExample::~CodeExample()
{
    delete [] m_buffer;
}

int wmain(int argc, wchar_t *argv[])
{
    CodeExample example(23);
    return 0;
}

```

The CodeExample class declaration defines m_buffer and then m_size, so it will initialize m_buffer and then m_size. The constructor is written with a different initialization order than the declaration order; it is written so that it appears that m_size is valid when m_buffer is initialized, and this is not the case. If the declaration order is changed then the code will work as expected.

3.13.2 Constructors

✅ **Do** minimal work in the constructor. Constructors should not do much work other than to capture the constructor parameters and set main data members. The cost of any other processing should be delayed until required.

✅ **You should** be explicit in the copy semantics for a class. Copy constructors and assignment operators are special methods – if you don't provide an implementation, then the compiler will provide a default implementation for you. If copying is not supported by the class semantics, explicitly disallow it by providing private, unimplemented copy constructor and assignment operators. For example:

```

class Example
{
private:
    Example(const Example&);
    Example& operator=(const Example&);
};

```

You should not provide implementations of these methods - this ensures that if they are ever accidentally used, a compiler error will be generated to alert you.

✅ **Do** define copy constructors as taking a 'const' reference type. For example, for a class T, the copy constructor should be defined as:

```

T(const T& other)

```

```
{
    ...
}
```

If the constructor was defined as “T(T& other)” or even “T(T& other, int value = 0)”, they would still be copy constructors. By standardizing on “const T&”, the constructor will work for both const and non-const values, with the added safety that const-ness brings.

✅ **Do** define all single parameter constructors, by default, with the ‘explicit’ keyword, so that they are not **conversion constructors**. For example,

```
class CodeExample
{
    int m_value;
public:
    explicit CodeExample(int value) :
        m_value(value)
    {
    }
};
```

❌ **Do not** provide conversion constructors unless the semantics of the class justify them.

3.13.3 Destructors

✅ **Do** use a destructor to centralize the resource cleanup of a class that is freed via delete. If resources are freed before destruction, make sure the fields are reset (e.g. set pointers to NULL) so that a destructor will not try to free them again.

✅ **Do** declare the destructor as "virtual" for classes that contain at least one other virtual function. If the class does not contain any virtual functions, then do not declare the destructor as virtual.

Here is the rationale behind the rule of using a virtual destructor if and only if the class has a virtual function. Assume class B derives from class A, and you have a pointer p which is of type A. p can actually hold an object of type A or B. If A and B have a virtual function F, then p->F() will call A::F if p points to an A object, or B::F if it points to a B object. You obviously need the matching destructor ~A or ~B, so you need the destructor to be virtual. But what if F is not virtual? Then, regardless of what p points to, you end up calling A::F. If p points to a B, then you’re treating the B as though it were an A. Therefore, you definitely don’t want to call B::F because the member functions are updating the state of an A, not the state of a B. The B destructor may fail if it touches state that only applies to a B. It’s this latter case where a virtual destructor creates the problem. Abusing virtual destructors is the source of a lot of C++ bugs.

3.13.4 Operators

❌ **Do not** overload operator&&, operator|| or operator,. Unlike the built-in &&, || or , operators the overloaded versions *cannot* be short-circuited, so the resulting behavior of using these operators typically isn’t what was expected.

❌ **You should not** overload operators unless the semantics of the class justify it.

❌ **Do not** change the semantics of the operators if you choose to overload them. For example, do not repurpose the '+' operator for performing subtraction.

❌ **You should not** implement [conversion operators](#) unless the semantics of the class justify them.

3.13.5 Function Overloading

❌ **Do not** arbitrarily varying parameter names in overloads. If a parameter in one overload represents the same input as a parameter in another overload, the parameters should have the same name. Parameters with the same name should appear in the same position in all overloads.

✅ **Do** make only the longest overload virtual (if extensibility is required). Shorter overloads should simply call through to a longer overload.

3.13.6 Virtual Functions

✅ **Do** use virtual functions to implement polymorphism

❌ **Do not** use virtual methods unless you really should because virtual functions have overhead of calling through the vtable.

✅ **You should** use the 'override' keyword when overriding a virtual function (Note: this is a Microsoft specific extension to C++). The override keyword will cause the compiler to ensure that the method prototype matches a virtual function in a base class. If a change is made to the prototype of a virtual function on a base class, then the compiler will generate errors for derived classes that need to be fixed.

For example:

```
class Example
{
protected:
    Example()
    {
    }

public:
    virtual ~Example()
    {
    }

    virtual int MeasureAccuracy();

private:
    Example(const Example&);
    Example& operator=(const Example&);
};

class ConcreteExample : public Example
```

```

{
public:
    ConcreteExample()
    {
    }

    ~ConcreteExample ()
    {
    }

    int MeasureAccuracy() override;
};

```

3.13.7 Abstract Classes

An abstract class provides a polymorphic base class and requires a derived class to provide implementation for virtual methods.

✅ **You can** use the 'abstract' keyword to identify an abstract class (Note: this is a Microsoft specific extension to C++).

✅ **Do** provide a protected constructor.

✅ **Do** identify abstract methods by making them pure virtual.

✅ **Do** provide a public, virtual destructor if you allow deletion via a pointer to the abstract class **or** a protected, non-virtual destructor to disallow deletion via a pointer to the abstract class.

✅ **Do** explicitly provide protected copy constructor and assignment operators **or** private unimplemented copy constructor and assignment operators – this will cause a compilation error if a user accidentally uses an abstract base class that results in pass-by-value behavior.

An example of an abstract class:

```

class Example abstract
{
protected:
    Example()
    {
    }

public:
    virtual ~Example()
    {
    }

    virtual int MeasureAccuracy() = 0;
};

```

```
private:
    Example(const Example&);
    Example& operator=(const Example&);
};
```

3.14 COM

3.14.1 COM Interfaces

☑ **Do** use IFACEMETHODIMP and IFACEMETHODIMP_ for method declarations in COM interfaces. These macros have replaced the usage of STDMETHODCALLTYPE and STDMETHODCALLTYPE as they add the __override SAL annotation.

For example:

```
class CNetDataObj : public IDataObject
{
public:
    // IDataObject
    IFACEMETHODIMP GetData(FORMATETC *pFmtEtc, STGMEDIUM *pmedium)

    ...

    IFACEMETHODIMP CNetDataObj::GetData(FORMATETC *pFmtEtc, STGMEDIUM *pmedium)
    {
        ...
    }
}
```

☑ **Do** order interface methods in your class definition in the same order they are declared in their definition. This is the order for IUnknown: QueryInterface()/AddRef()/Release().

3.14.2 COM Interface ID

__uuidof() is a compiler supported feature that produces a single GUID value that might be associated with a type. That type might be an interface pointer, class. The GUID is associated with that type using __declspec(uuidof("<guid value>")).

Avoid using either when you can, use IID_PPV_ARGS() or the templated QueryInterface()

Use __uuidof(var) when you need the IID of an interface pointer rather than hard coding the knowledge of the IID of that variable. This is similar to the use of sizeof(var) vs sizeof(TYPE_OF_VAR). Note you can type "sizeof var", no parentheses needed. For example,

```
CoMarshalInterThreadInterfaceInStream(__uuidof(psia), psia, &pstm);
```

When faced with using IID_ISomeInterface vs __uuidof(ISomeInterface) prefer the former since it is more concise. Same applies to CLSID_SomeCoClass vs __uuidof(SomeCoClass). One exception to this rule is explicitly referring to the IID or CLSID requires linking to it via a .lib. If none is supplied then the __uuidof() is preferred to having to define that value in your code.

3.14.3 COM Classes

✓ **Do** declare private destructors (or protected if you expect people to derive from you) for classes that implement COM objects that are allocated on the heap. This avoids clients mistakenly calling “delete pObj”, something that should only happen when the ref count of the object goes to zero.

✓ **Do** initialize the m_cRef to 1 on construction for classes that implement COM object. (Note: ATL uses a different pattern of ref initiated to 0 expecting the use of smart pointers whose assignment increments the value to 1). This makes it impossible to be in the state where the class exists but cannot be released.

✓ **Do** return an HRESULT from every COM method (except AddRef and Release).

3.15 Allocations

✓ **Do** ensure that all allocated memory is freed using the same mechanisms. Objects allocated using ‘new’ should be freed with ‘delete’. For example:

```
Engine *pEngine = new Engine();
pEngine->Process();
delete pEngine;
```

Allocations made using ‘vector new’ should be freed using ‘vector delete’. For example:

```
wchar_t *pszBuffer = new wchar_t[MAX_PATH];
SomeMethod(pszBuffer);
delete [] pszBuffer;
```

✓ **Do** understand the allocations within your code base to ensure that they are freed correctly.

3.15.1 Smart Pointers

✓ **You should** use RAI (Resource Allocation Is Initialization) constructs to help track allocations – using smart pointers, for example. The previous two examples written using ATL’s ‘smart pointer’ classes would be:

```
{
    CAutoPtr<Engine> spEngine(new Engine());
    spEngine->Process();
}

{
    CAutoVectorPtr<wchar_t> spBuffer();
    spBuffer.Allocate(MAX_PATH);
    SomeMethod(spBuffer);
}
```

✗ **Do not** enable ATL in your project just because you want to use CAutoPtr and CAutoVectorPtr.

3.16 Errors and Exceptions

✓ **Do** prefer error code return values to exception handling in most cases, for simplicity. Always use error codes from DLL-exported APIs or methods.

3.16.1 Errors

✓ **Do** check return values for function calls and handle errors appropriately. When detecting an error, print the error message as early as possible in console applications, and handle the error. For example,

```
// Function returns HRESULT.
IShellLibrary* pShellLib = NULL;
HRESULT hr = SHCreateLibrary(IID_PPV_ARGS(&pShellLib));
if (FAILED(hr))
{
    wprintf(L"SHCreateLibrary failed w/err 0x%08lx\n", hr);
    goto Cleanup;
}

// Function returns TRUE/FALSE and sets the Win32 last error.
DWORD dwError = ERROR_SUCCESS;
HANDLE hToken = NULL;
if (!OpenProcessToken(GetCurrentProcess(), TOKEN_QUERY | TOKEN_DUPLICATE,
    &hToken))
{
    dwError = GetLastError();
    wprintf(L"OpenProcessToken failed w/err 0x%08lx\n", dwError);
    goto Cleanup;
}
```

3.16.2 Exceptions

Exceptions are a feature of C++ that requires a good understanding before they can be used appropriately. Before consuming code that uses native C++ exceptions, make sure that you understand the implications of consuming that code.

Native C++ exceptions are a powerful feature of the language, and can reduce the complexity of code, and reduce the amount of code that is written and maintained.

✓ **Do** throw exceptions by value and catch exceptions by reference. For example,

```
void ProcessItem(const Item& item)
{
    try
    {
        if (/* some test failed */)
        {
            throw _com_error(E_FAIL);
        }
    }
}
```

```

    }
    catch(_com_error& comError)
    {
        // Process comError
        //
    }
}

```

✅ When re-throwing exceptions **do** re-throw exceptions using “throw;” instead of “throw <caught exception>”. For example,

Good:

```

void ProcessItem(const Item& item)
{
    try
    {
        Item->Process();
    }
    catch(ItemException& itemException)
    {
        wcout << L"An error occurred."
        throw;
    }
}

```

Bad:

```

void ProcessItem(const Item& item)
{
    try
    {
        Item->Process();
    }
    catch(ItemException& itemException)
    {
        wcout << L"An error occurred."
        throw itemException;
    }
}

```

❌ **Do not** allow exceptions to be thrown out of destructors.

❌ **Do not** use “catch(...)”. General exceptions should not be caught. You should catch a more specific exception, or re-throw the general exception as the last statement in the catch block. There are cases when swallowing errors in applications is acceptable, but such cases are rare. Only catch the specific exceptions that the function knows how to handle. All others must be passed unhandled.

❌ **Do not** use exceptions for control flow. Except for system failures and operations with potential race conditions, you should write code that does not throw exceptions. For example, you can check preconditions before calling a method that may fail and throw exceptions. For example,

```
if (IsWritable(list))
{
    WriteList(list);
}
```

✅ **Do** make sure that you understand the exceptions that may be thrown from code that you take a dependency on, and ensure that the exceptions aren't unintentionally propagated to the consumers of your API. For example, STL and ATL can throw native C++ exceptions in certain scenarios – understand those scenarios and ensure that the appropriate exceptions are handled in your code to prevent propagation out.

3.17 Resource Cleanup

Dynamically allocated memory / resources must be appropriately cleaned up before you exit from a function to avoid memory / resource leaks. Proper resource cleanup solution is particularly important when an error occurs in the middle of the function. Here are five commonly seen patterns of code to clean up resources in a function.

Pattern	Example	Analysis
goto Cleanup	<pre>HANDLE hToken = NULL; PVOID pMem = NULL; if (!OpenProcessToken(GetCurrentProcess(), TOKEN_QUERY, &hToken)) { ReportError(GetLastError()); goto Cleanup; } pMem = LocalAlloc(LPTR, 10); if (pMem == NULL) { ReportError(GetLastError()); goto Cleanup; } ... Cleanup: if (hToken) { CloseHandle(hToken); hToken = NULL; } if (pMem) { LocalFree(pMem); pMem = NULL; }</pre>	<p>If you are absolutely sure that the code does not throw an exception, "goto Cleanup" should be the best choice. It is faster than "__try/__finally", is easier to implement than "Early return with RAII wrapper", and is portal to C.</p>
__try / __finally	<pre>HANDLE hToken = NULL; PVOID pMem = NULL; __try {</pre>	<p>__try/__finally is not portable to other systems, hampers the optimizer, and is surprisingly more</p>

	<pre> if (!OpenProcessToken(GetCurrentProcess(), TOKEN_QUERY, &hToken)) { ReportError(GetLastError()); __leave; } pMem = LocalAlloc(LPTR, 10); if (pMem == NULL) { ReportError(GetLastError()); __leave; } ... } __finally { if (hToken) { CloseHandle(hToken); hToken = NULL; } if (pMem) { LocalFree(pMem); pMem = NULL; } } </pre>	<p>expensive than goto and early return. <code>__try / __finally</code> inhibits a large set of optimizations because the compiler must assume that something bad can happen at any time (in the middle of an expression, or inside a function you called). In contrast, "goto" lets the compiler assume that bad things happen only when you do a "goto".</p>
Nested if	<pre> HANDLE hToken = NULL; PVOID pMem = NULL; if (OpenProcessToken(GetCurrentProcess(), TOKEN_QUERY, &hToken)) { pMem = LocalAlloc(LPTR, 10); if (pMem) { ... LocalFree(pMem); pMem = NULL; } else { ReportError(GetLastError()); } CloseHandle(hToken); hToken = NULL; } else { ReportError(GetLastError()); } </pre>	<p>Nested if is often the worst choice. You quickly run out of horizontal space. It is harder to read, harder to maintain.</p>
Early return with repeating cleanup	<pre> DWORD dwError = ERROR_SUCCESS; HANDLE hToken = NULL; PVOID pMem = NULL; if (!OpenProcessToken(GetCurrentProcess(), TOKEN_QUERY TOKEN_DUPLICATE, &hToken)) { </pre>	<p>Early return is bad in C because you end up repeating the cleanup code. If the function has no cleanup or it is an extremely small function, then early return can be ok.</p>

	<pre> ReportError(GetLastError()); return FALSE; } pMem = LocalAlloc(LPTR, 10); if (pMem == NULL) { ReportError(GetLastError()); CloseHandle(hToken); hToken = NULL; return FALSE; } CloseHandle(hToken); LocalFree(pMem); return TRUE; </pre>	
Early return with RAII wrapper	<pre> namespace WinRAII { class AutoFreeObjHandle { public: explicit AutoFreeObjHandle(HANDLE h): m_hObj(h) { ; } ~AutoFreeObjHandle() { Close(); } void Close(void) { if (m_hObj) { CloseHandle(m_hObj); m_jObj = NULL; } } HANDLE Get(void) const { return (m_hObj); } void Set(HANDLE h) { m_hObj = h; } private: HANDLE m_hObj; AutoFreeObjHandle(void); AutoFreeObjHandle(const AutoFreeObjHandle &); AutoFreeObjHandle & operator = (const AutoFreeObjHandle &); } } WinRAII::AutoFreeObjHandle afToken(NULL); if (!OpenThreadToken (GetCurrentThread(), TOKEN_QUERY, TRUE, &hToken)) { ReportError(GetLastError()); return FALSE; } </pre>	<p>If cleanup is in destructors then early return is ok. Because C++ exception handling doesn't have a finally or a C# using, you need an RAII-style wrapper class for each resource type. The compiler generates code to call the destructors of all stack-based objects (the wrappers) when the function exits. This makes them equivalent to a <code>__finally</code> if you throw.</p>

3.18 Control Flow

3.18.1 Early Returns

❗ **You should not** early return in most functions. Early returns are acceptable in some situations, but they should be avoided. Ideally functions will have a single return point at the bottom that all execution leads to.

Acceptable situations for early returns are:

- Parameter validation done at the very beginning of a function
- Extremely small functions (like assessors to member variable state)

3.18.2 Goto

❌ **Do not** use 'goto' statements in place of structured control flow in attempts to “optimize” runtime performance. Doing so leads to code which is very hard to understand, debug, and verify correct.

✅ **You can** use goto in a structured manner by always jumping forward and by implementing a consist set of jumps related to a single purpose such as jumping out of a series of resource allocations into cleanup code when one resource cannot be allocated. Such use of goto can reduce deep levels of nesting and make error handling much easier to see and verify. For example:

```

BOOL IsElevatedAdministrator(HANDLE hInputToken)
{
    BOOL fIsAdmin = FALSE;
    HANDLE hTokenToCheck = NULL;

    // If caller supplies a token, duplicate it to an impersonation token
    // because CheckTokenMembership requires an impersonation token.
    if (hInputToken)
    {
        if (!DuplicateToken(hInputToken, SecurityIdentification, &hTokenToCheck))
        {
            goto CLEANUP;
        }
    }

    DWORD sidLen = SECURITY_MAX_SID_SIZE;
    BYTE localAdminsGroupSid[SECURITY_MAX_SID_SIZE];

    if (!CreateWellKnownSid(WinBuiltinAdministratorsSid, NULL,
        localAdminsGroupSid, &sidLen))
    {
        goto CLEANUP;
    }

    // Now, determine if the user is an admin
    if (CheckTokenMembership(hTokenToCheck, localAdminsGroupSid, &fIsAdmin))
    {
        // lastErr = ERROR_SUCCESS;
    }

CLEANUP:
    // Close the impersonation token only if we opened it.
    if (hTokenToCheck)

```

```

    {
        CloseHandle(hTokenToCheck);
        hTokenToCheck = NULL;
    }

    return (fIsAdmin);
}

```

Logically, this code is structured as four chunks (shaded below). The first two chunks try to allocate resources; if they succeed, control falls through to the next chunk. If one fails, control goes to the cleanup code. The third chunk is the final logic which determines the final result of the function, then falls through to the cleanup code. This is a structured use of goto because all gotos go forward and each one is used in a consistent way for a single purpose. The resulting code is shorter, easier to read, and easier to verify than the nested equivalent. This is a good use of goto.

```

BOOL IsElevatedAdministrator(HANDLE hInputToken)
{
    BOOL fIsAdmin = FALSE;
    HANDLE hTokenToCheck = NULL;
    // If caller supplies a token, duplicate it to an impersonation token
    // because CheckTokenMembership requires an impersonation token.
    if (hInputToken)
    {
        if (!DuplicateToken(hInputToken, SecurityIdentification, &hTokenToCheck))
        {
            goto CLEANUP;
        }
    }
}

```

```

DWORD sidLen = SECURITY_MAX_SID_SIZE;
BYTE localAdminsGroupSid[SECURITY_MAX_SID_SIZE];

if (!CreateWellKnownSid(WinBuiltinAdministratorsSid, NULL,
    localAdminsGroupSid, &sidLen))
{
    goto CLEANUP;
}

```

```

// Now, determine if the user is an admin
if (CheckTokenMembership(hTokenToCheck, localAdminsGroupSid, &fIsAdmin))
{
    // lastErr = ERROR_SUCCESS;
}

```

```

CLEANUP:
    // Close the impersonation token only if we opened it.
    if (hTokenToCheck)

```



```
{  
    CloseHandle(hTokenToCheck);  
    hTokenToCheck = NULL;  
}  
  
return (fIsAdmin);  
}
```


4 .NET Coding Standards


These coding standards can be applied to C# and VB.NET.

4.1 Design Guidelines for Developing Class Libraries

The [Design Guidelines for Developing Class Libraries](#) document on MSDN is a fairly thorough discussion of how to write managed code. The information in this section highlights some important standards and lists the All-In-One Code Framework code samples' exceptions to the guidelines. Therefore, you had better read the two documents side by side.

4.2 Files and Structure

 **Do not** have more than one public type in a source file, unless they differ only in the number of generic parameters or one is nested in the other. Multiple internal types in one file are allowed.

 **Do** name the source file with the name of the public type it contains. For example, MainForm class should be in MainForm.cs file and List<T> class should be in List.cs file.


4.3 Assembly Properties


The assembly should contain the appropriate property values describing its name, copyright, and so on.


Standard	Example
Set Copyright to Copyright © Microsoft Corporation 2010	<code>[assembly: AssemblyCopyright("Copyright © Microsoft Corporation 2010")]</code>
Set AssemblyCompany to Microsoft Corporation	<code>[assembly: AssemblyCompany("Microsoft Corporation")]</code>
Set both AssemblyTitle and AssemblyProduct to the current sample name	<code>[assembly: AssemblyTitle("CSNamedPipeClient")]</code> <code>[assembly: AssemblyProduct("CSNamedPipeClient")]</code>

4.4 Naming Conventions

4.4.1 General Naming Conventions










 **Do** use meaning names for various types, functions, variables, constructs and types.

 **You should not** use of shortenings or contractions as parts of identifier names. For example, use "GetWindow" rather than "GetWin". For functions of common types, thread procs, window procedures, dialog procedures use the common suffixes for these "ThreadProc", "DialogProc", "WndProc".

 **Do not** use underscores, hyphens, or any other non-alphanumeric characters.

4.4.2 Capitalization Naming Rules for Identifiers

The following table describes the capitalization and naming rules for different types of identifiers.

Identifier	Casing	Naming Structure	Example
Class, Structure	PascalCasing	Noun	<pre>public class ComplexNumber {...} public struct ComplexStruct {...}</pre>
Namespace	PascalCasing	Noun  Do not use the same name for a namespace and a type in that namespace.	<pre>namespace Microsoft.Sample.Windows7</pre>
Enumeration	PascalCasing	Noun  Do name flag enums with plural nouns or noun phrases and simple enums with singular nouns or noun phrases.	<pre>[Flags] public enum ConsoleModifiers { Alt, Control }</pre>
Method	PascalCasing	Verb or Verb phrase	<pre>public void Print() {...} public void ProcessItem() {...}</pre>
Public Property	PascalCasing	Noun or Adjective  Do name collection proprieties with a plural phrase describing the items in the collection, as opposed to a singular phrase followed by “List” or “Collection”.  Do name Boolean proprieties with an affirmative phrase (CanSeek instead of CantSeek). Optionally, you can also prefix Boolean properties with “Is,” “Can,” or “Has” but only where it adds value.	<pre>public string CustomerName public ICollection Items public bool CanRead</pre>
Non-public Field	camelCasing or _camelCasing	Noun or Adjective.  Do be consistent in a code sample when you use the '_' prefix.	<pre>private string name; private string _name;</pre>
Event	PascalCasing	Verb or Verb phrase  Do give events names with a concept of before and after, using the present and past tense.  Do not use “Before” or “After” prefixes or postfixes to indicate pre and post events.	<pre>// A close event that is raised after the window is closed. public event WindowClosed // A close event that is raised before a window is closed. public event WindowClosing</pre>
Delegate	PascalCasing	 Do add the suffix ‘EventHandler’ to names of delegates that are used in events.  Do add the suffix ‘Callback’ to names of delegates other than those used as event handlers.	<pre>public delegate WindowClosedEventHandler</pre>

		❌ Do not add the suffix “Delegate” to a delegate.	
Interface	PascalCasing 'I' prefix	Noun	<code>public interface IDictionary</code>
Constant	PascalCasing for publicly visible; camelCasing for internally visible; All capital only for abbreviation of one or two chars long.	Noun	<code>public const string MessageText = "A"; private const string messageText = "B"; public const double PI = 3.14159...;</code>
Parameter, Variable	camelCasing	Noun	<code>int customerID;</code>
Generic Type Parameter	PascalCasing 'T' prefix	Noun ✅ Do name generic type parameters with descriptive names, unless a single-letter name is completely self-explanatory and a descriptive name would not add value. ✅ Do prefix descriptive type parameter names with T. ✅ You should using T as the type parameter name for types with one single-letter type parameter.	<code>T, TItem, TPolicy</code>
Resource	PascalCasing	Noun ✅ Do provide descriptive rather than short identifiers. Keep them concise where possible, but do not sacrifice readability for space. ✅ Do use only alphanumeric characters and underscores in naming resources.	<code>ArgumentExceptionInvalidName</code>

4.4.3 Hungarian Notation

❌ **Do not** use Hungarian notation (i.e., do not encode the type of a variable in its name) in .NET.

4.4.4 UI Control Naming Conventions

UI controls would use the following prefixes. The primary purpose was to make code more readable.

Control Type	Prefix
--------------	--------

Button	btn
CheckBox	chk
CheckedListBox	lst
ComboBox	cmb
ContextMenu	mnu
DataGrid	dg
DateTimePicker	dtg
Form	suffix: XXXForm
GroupBox	grp
ImageList	iml
Label	lb
ListBox	lst
ListView	lvw
Menu	mnu
MenuItem	mnu
NotificationIcon	nfy
Panel	pnl
PictureBox	pct
ProgressBar	prg
RadioButton	rad
Splitter	spl
StatusBar	sts
TabControl	tab
TabPage	tab
TextBox	tb
Timer	tmr
TreeView	tvw

For example, for the “File | Save” menu option, the “Save” MenuItem would be called “mnuFileSave”.

4.5 Constants

☒ **Do** use constant fields for constants that will never change. The compiler burns the values of const fields directly into calling code. Therefore const values can never be changed without the risk of breaking compatibility.

```
public class Int32
{
    public const int MaxValue = 0x7fffffff;
```

```

        public const int MinValue = unchecked((int)0x80000000);
    }

    Public Class Int32
        Public Const MaxValue As Integer = &H7FFFFFFF
        Public Const MinValue As Integer = &H80000000
    End Class

```

✅ **Do** use public static (shared) readonly fields for predefined object instances. If there are predefined instances of the type, declare them as public readonly static fields of the type itself. For example,

```

public class ShellFolder
{
    public static readonly ShellFolder ProgramData = new ShellFolder("ProgramData");
    public static readonly ShellFolder ProgramFiles = new ShellFolder("ProgramData");
    ...
}

Public Class ShellFolder
    Public Shared ReadOnly ProgramData As New ShellFolder("ProgramData")
    Public Shared ReadOnly ProgramFiles As New ShellFolder("ProgramFiles")
    ...
End Class

```

4.6 Strings

❌ **Do not** use the '+' operator (or '&' in VB.NET) to concatenate many strings. Instead, you should use `StringBuilder` for concatenation. However, **do** use the '+' operator (or '&' in VB.NET) to concatenate small numbers of strings.

Good:

```

StringBuilder sb = new StringBuilder();
for (int i = 0; i < 10; i++)
{
    sb.Append(i.ToString());
}

```

Bad:

```

string str = string.Empty;
for (int i = 0; i < 10; i++)
{
    str += i.ToString();
}

```

✅ **Do** use overloads that explicitly specify the string comparison rules for string operations. Typically, this involves calling a method overload that has a parameter of type `StringComparison`.

✓ **Do** use [StringComparison.Ordinal](#) or [StringComparison.OrdinalIgnoreCase](#) for comparisons as your safe default for culture-agnostic string matching, and for better performance.

✓ **Do** use string operations that are based on [StringComparison.CurrentCulture](#) when you display output to the user.

✓ **Do** use the non-linguistic [StringComparison.Ordinal](#) or [StringComparison.OrdinalIgnoreCase](#) values instead of string operations based on [CultureInfo.InvariantCulture](#) when the comparison is linguistically irrelevant (symbolic, for example). Do not use string operations based on [StringComparison.InvariantCulture](#) in most cases. One of the few exceptions is when you are persisting linguistically meaningful but culturally agnostic data.

✓ **Do** use an overload of the [String.Equals](#) method to test whether two strings are equal. For example, to test if two strings are equal ignoring the case,

```
if (str1.Equals(str2, StringComparison.OrdinalIgnoreCase))

If (str1.Equals(str2, StringComparison.OrdinalIgnoreCase)) Then
```

✗ **Do not** use an overload of the [String.Compare](#) or [CompareTo](#) method and test for a return value of zero to determine whether two strings are equal. They are used to sort strings, not to check for equality.

✓ **Do** use the [String.ToUpperInvariant](#) method instead of the [String.ToLowerInvariant](#) method when you normalize strings for comparison.

4.7 Arrays and Collections

✓ **You should** use arrays in low-level functions to minimize memory consumption and maximize performance. In public interfaces, do prefer collections over arrays.

Collections provide more control over contents, can evolve over time, and are more usable. In addition, using arrays for read-only scenarios is discouraged as the cost of cloning the array is prohibitive.

However, if you are targeting more skilled developers and usability is less of a concern, it might be better to use arrays for read-write scenarios. Arrays have a smaller memory footprint, which helps reduce the working set, and access to elements in an array is faster as it is optimized by the runtime.

✗ **Do not** use read-only array fields. The field itself is read-only and can't be changed, but elements in the array can be changed. This example demonstrates the pitfalls of using read-only array fields:

```
Bad:
public static readonly char[] InvalidPathChars = { '\\', '<', '>', '|' };
```

This allows callers to change the values in the array as follows:

```
InvalidPathChars[0] = 'A';
```

Instead, you can use either a read-only collection (only if the items are immutable) or clone the array before returning it. However, the cost of cloning the array may be prohibitive.

```
public static ReadOnlyCollection<char> GetInvalidPathChars()
{
    return Array.AsReadOnly(badChars);
}

public static char[] GetInvalidPathChars()
{
    return (char[])badChars.Clone();
}
```

✅ **You should** use jagged arrays instead of multidimensional arrays. A jagged array is an array with elements that are also arrays. The arrays that make up the elements can be of different sizes, leading to less wasted space for some sets of data (e.g., sparse matrix), as compared to multidimensional arrays. Furthermore, the CLR optimizes index operations on jagged arrays, so they might exhibit better runtime performance in some scenarios.

```
// Jagged arrays
int[][] jaggedArray =
{
    new int[] {1, 2, 3, 4},
    new int[] {5, 6, 7},
    new int[] {8},
    new int[] {9}
};

Dim jaggedArray As Integer()() = New Integer()() _
{ _
    New Integer() {1, 2, 3, 4}, _
    New Integer() {5, 6, 7}, _
    New Integer() {8}, _
    New Integer() {9} _
}

// Multidimensional arrays
int [,] multiDimArray =
{
    {1, 2, 3, 4},
    {5, 6, 7, 0},
    {8, 0, 0, 0},
    {9, 0, 0, 0}
};

Dim multiDimArray(,) As Integer = _
{ _
```



```

        {1, 2, 3, 4}, _
        {5, 6, 7, 0}, _
        {8, 0, 0, 0}, _
        {9, 0, 0, 0} _
    }

```

✅ **Do** use `Collection<T>` or a subclass of `Collection<T>` for properties or return values representing read/write collections, and use `ReadOnlyCollection<T>` or a subclass of `ReadOnlyCollection<T>` for properties or return values representing read-only collections.

✅ **You should** reconsider the use of `ArrayList` because any objects added into the `ArrayList` are added as `System.Object` and when retrieving values back from the `arraylist`, these objects are to be unboxed to return the actual value type. So it is recommended to use the custom typed collections instead of `ArrayList`. For example, .NET provides a strongly typed collection class for `String` in `System.Collection.Specialized`, namely `StringCollection`.

✅ **You should** reconsider the use of `Hashtable`. Instead, try other dictionary such as `StringDictionary`, `NameValueCollection`, `HybridCollection`. `Hashtable` can be used if less number of values is stored.

✅ When you are creating a collection type, **you should** implement `IEnumerable` so that the collection can be used with LINQ to Objects.

❌ **Do not** implement both `IEnumerator<T>` and `IEnumerable<T>` on the same type. The same applies to the nongeneric interfaces `IEnumerator` and `IEnumerable`. In other words, a type should be either a collection or an enumerator, but not both.

❌ **Do not** return a null reference for `Array` or `Collection`. Null can be difficult to understand in this context. For example, a user might assume that the following code will work. Return an empty array or collection instead of a null reference.

```

int[] arr = SomeOtherFunc();
foreach (int v in arr)
{
    ...
}

```

4.8 Structures

✅ **Do** ensure that a state where all instance data is set to zero, false, or null (as appropriate) is valid. This prevents accidental creation of invalid instances when an array of the structs is created.

✅ **Do** implement `IEquatable<T>` on value types. The `Object.Equals` method on value types causes boxing and its default implementation is not very efficient, as it uses reflection. `IEquatable<T>.Equals` can have much better performance and can be implemented such that it will not cause boxing.

4.8.1 Structures vs. Classes

❌ **Do not** define a struct unless the type has all of the following characteristics:

- It logically represents a single value, similar to primitive types (int, double, etc.).
- It has an instance size fewer than 16 bytes.
- It is immutable.
- It will not have to be boxed frequently.

In all other cases, you should define your types as classes instead of structs.

4.9 Classes

✓ **Do** use inheritance to express “is a” relationships such as “cat is an animal”.

✓ **Do** use interfaces such as IDisposable to express “can do” relationships such as using “objects of this class can be disposed”.

4.9.1 Fields

✗ **Do not** provide instance fields that are public or protected. Public and protected fields do not version well and are not protected by code access security demands. Instead of using publicly visible fields, use private fields and expose them through properties.

✓ **Do** use public static read-only fields for predefined object instances.

✓ **Do** use constant fields for constants that will never change.

✗ **Do not** assign instances of mutable types to read-only fields.

4.9.2 Properties

✓ **Do** create read-only properties if the caller should not be able to change the value of the property.

✗ **Do not** provide set-only properties. If the property getter cannot be provided, use a method to implement the functionality instead. The method name should begin with Set followed by what would have been the property name.

✓ **Do** provide sensible default values for all properties, ensuring that the defaults do not result in a security hole or an extremely inefficient design.

✗ **You should not** throw exceptions from property getters. Property getters should be simple operations without any preconditions. If a getter might throw an exception, consider redesigning the property to be a method. This recommendation does not apply to indexers. Indexers can throw exceptions because of invalid arguments. It is valid and acceptable to throw exceptions from a property setter.

4.9.3 Constructors

✓ **Do** minimal work in the constructor. Constructors should not do much work other than to capture the constructor parameters and set main properties. The cost of any other processing should be delayed until required.

✓ **Do** throw exceptions from instance constructors if appropriate.

✓ **Do** explicitly declare the public default constructor in classes, if such a constructor is required. Even though some compilers automatically add a default constructor to your class, adding it explicitly makes code maintenance easier. It also ensures the default constructor remains defined even if the compiler stops emitting it because you add a constructor that takes parameters.

✗ **Do not** call virtual members on an object inside its constructors. Calling a virtual member causes the most-derived override to be called regardless of whether the constructor for the type that defines the most-derived override has been called.

4.9.4 Methods

✓ **Do** place all out parameters after all of the pass-by-value and ref parameters (excluding parameter arrays), even if this results in an inconsistency in parameter ordering between overloads.

✓ **Do** validate arguments passed to public, protected, or explicitly implemented members. Throw `System.ArgumentException`, or one of its subclasses, if the validation fails: If a null argument is passed and the member does not support null arguments, throw `ArgumentNullException`. If the value of an argument is outside the allowable range of values as defined by the invoked method, throw `ArgumentOutOfRangeException`.

4.9.5 Events

✓ **Do** be prepared for arbitrary code executing in the event-handling method. Consider placing the code where the event is raised in a try-catch block to prevent program termination due to unhandled exceptions thrown from the event handlers.

✗ **Do not** use events in performance sensitive APIs. While events are easier for many developers to understand and use, they are less desirable than Virtual Members from a performance and memory consumption perspective.

4.9.6 Member Overloading

✓ **Do** use member overloading rather than defining members with default arguments. Default arguments are not CLS-compliant and cannot be used from some languages. There is also a versioning issue in members with default arguments. Imagine version 1 of a method that sets an optional parameter to 123. When compiling code that calls this method without specifying the optional parameter, the compiler will embed the default value (123) into the code at the call site. Now, if version 2 of the method changes the optional parameter to 863, then, if the calling code is not recompiled, it will call version 2 of the method passing in 123 (version 1's default, not version 2's default).

Good:

```
Public Overloads Sub Rotate(ByVal data As Matrix)
    Rotate(data, 180)
End Sub
```

```
Public Overloads Sub Rotate(ByVal data As Matrix, ByVal degrees As Integer)
    ' Do rotation here
End Sub
```

Bad:

```
Public Sub Rotate(ByVal data As Matrix, Optional ByVal degrees As Integer = 180)
    ' Do rotation here
End Sub
```

❌ **Do not** arbitrarily vary parameter names in overloads. If a parameter in one overload represents the same input as a parameter in another overload, the parameters should have the same name. Parameters with the same name should appear in the same position in all overloads.

✅ **Do** make only the longest overload virtual (if extensibility is required). Shorter overloads should simply call through to a longer overload.

4.9.7 Interface Members

❌ **You should not** implement interface members explicitly without having a strong reason to do so. Explicitly implemented members can be confusing to developers because they don't appear in the list of public members and they can also cause unnecessary boxing of value types.

✅ **You should** implement interface members explicitly, if the members are intended to be called only through the interface.

4.9.8 Virtual Members

Virtual members perform better than callbacks and events, but do not perform better than non-virtual methods.

❌ **Do not** make members virtual unless you have a good reason to do so and you are aware of all the costs related to designing, testing, and maintaining virtual members.

✅ **You should** prefer protected accessibility over public accessibility for virtual members. Public members should provide extensibility (if required) by calling into a protected virtual member.

4.9.9 Static Classes

✅ **Do** use static classes sparingly. Static classes should be used only as supporting classes for the object-oriented core of the framework.

4.9.10 Abstract Classes

❌ **Do not** define public or protected-internal constructors in abstract types.

✅ **Do** define a protected or an internal constructor on abstract classes.

A protected constructor is more common and simply allows the base class to do its own initialization when subtypes are created.

```
public abstract class Claim
{
    protected Claim()
    {
    }
}
```

```

        ...
    }
}

```

An internal constructor can be used to limit concrete implementations of the abstract class to the assembly defining the class.

```

public abstract class Claim
{
    internal Claim()
    {
        ...
    }
}

```

4.10 Namespaces

✓ **Do** use the default namespaces of projects created by Visual Studio in All-In-One Code Framework code samples. It is not necessary to rename the namespace to the form of Microsoft.Sample.TechnologyName.

4.11 Errors and Exceptions

4.11.1 Exception Throwing

✓ **Do** report execution failures by throwing exceptions. Exceptions are the primary means of reporting errors in frameworks. If a member cannot successfully do what it is designed to do, it should be considered an execution failure and an exception should be thrown. **Do not** return error codes.

✓ **Do** throw the most specific (the most derived) exception that makes sense. For example, throw `ArgumentNullException` and not its base type `ArgumentException` if a null argument is passed. Throwing `System.Exception` as well as catching `System.Exception` are nearly always the wrong thing to do.

✗ **Do not** use exceptions for the normal flow of control, if possible. Except for system failures and operations with potential race conditions, you should write code that does not throw exceptions. For example, you can check preconditions before calling a method that may fail and throw exceptions. For example,

```

// C# sample:
if (collection != null && !collection.IsReadOnly)
{
    collection.Add(additionalNumber);
}

' VB.NET sample:
If ((Not collection Is Nothing) And (Not collection.IsReadOnly)) Then
    collection.Add(additionalNumber)
End If

```

❌ **Do not** throw exceptions from exception filter blocks. When an exception filter raises an exception, the exception is caught by the CLR, and the filter returns false. This behavior is indistinguishable from the filter executing and returning false explicitly and is therefore very difficult to debug.

```
' VB.NET sample
' This is bad design. The exception filter (When clause)
' may throw an exception when the InnerException property
' returns null
Try
    ...
Catch e As ArgumentException _
    When e.InnerException.Message.StartsWith("File")
    ...
End Try
```

❌ **Do not** explicitly throw exceptions from finally blocks. Implicitly thrown exceptions resulting from calling methods that throw are acceptable.

4.11.2 Exception Handling

❌ **You should not** swallow errors by catching nonspecific exceptions, such as `System.Exception`, `System.SystemException`, and so on in .NET code. Do catch only specific errors that the code knows how to handle. You should catch a more specific exception, or re-throw the general exception as the last statement in the catch block. There are cases when swallowing errors in applications is acceptable, but such cases are rare.

```
Good:
// C# sample:
try
{
    ...
}
catch (System.NullReferenceException exc)
{
    ...
}
catch (System.ArgumentOutOfRangeException exc)
{
    ...
}
catch (System.InvalidCastException exc)
{
    ...
}

' VB.NET sample:
Try
    ...
Catch exc As System.NullReferenceException
```

```

    ...
Catch exc As System.ArgumentOutOfRangeException
    ...
Catch exc As System.InvalidCastException
    ...
End Try

```

Bad:

```

// C# sample:
try
{
    ...
}
catch (Exception ex)
{
    ...
}

' VB.NET sample:
Try
    ...
Catch ex As Exception
    ...
End Try

```

☒ **Do prefer using an empty throw when catching and re-throwing an exception. This is the best way to preserve the exception call stack.**

Good:

```

// C# sample:
try
{
    ... // Do some reading with the file
}
catch
{
    file.Position = position; // Unwind on failure
    throw; // Rethrow
}

' VB.NET sample:
Try
    ... ' Do some reading with the file
Catch ex As Exception
    file.Position = position ' Unwind on failure
    Throw ' Rethrow
End Try

```

Bad:

```
// C# sample:
try
{
    ... // Do some reading with the file
}
catch (Exception ex)
{
    file.Position = position; // Unwind on failure
    throw ex; // Rethrow
}

' VB.NET sample:
Try
    ... ' Do some reading with the file
Catch ex As Exception
    file.Position = position ' Unwind on failure
    Throw ex ' Rethrow
End Try
```

4.12 Resource Cleanup

❌ **Do not** force garbage collections with GC.Collect.

4.12.1 Try-finally Block

✅ **Do** use try-finally blocks for cleanup code and try-catch blocks for error recovery code. **Do not** use catch blocks for cleanup code. Usually, the cleanup logic rolls back resource (particularly, native resource) allocations. For example,

```
// C# sample:
FileStream stream = null;
try
{
    stream = new FileStream(...);
    ...
}
finally
{
    if (stream != null)
    {
        stream.Close();
    }
}

' VB.NET sample:
Dim stream As FileStream = Nothing
```



```

Try
    stream = New FileStream(...)
    ...
Catch ex As Exception
    If (stream IsNot Nothing) Then
        stream.Close()
    End If
End Try

```

C# and VB.NET provide the using statement that can be used instead of plain try-finally to clean up objects implementing the IDisposable interface.

```

// C# sample:
using (FileStream stream = new FileStream(...))
{
    ...
}

' VB.NET sample:
Using stream As New FileStream(...)
    ...
End Using

```

Many language constructs emit try-finally blocks automatically for you. Examples are C#/VB's using statement, C#'s lock statement, VB's SyncLock statement, C#'s foreach statement, and VB's For Each statement.

4.12.2 Basic Dispose Pattern

The basic implementation of the pattern involves implementing the System.IDisposable interface and declaring the Dispose(bool) method that implements all resource cleanup logic to be shared between the Dispose method and the optional finalizer. Please note that this section does not discuss providing a finalizer. Finalizable types are extensions to this basic pattern and are discussed in the next section. The following example shows a simple implementation of the basic pattern:

```

// C# sample:
public class DisposableResourceHolder : IDisposable
{
    private bool disposed = false;
    private SafeHandle resource; // Handle to a resource

    public DisposableResourceHolder()
    {
        this.resource = ... // Allocates the native resource
    }

    public void DoSomething()
    {
        if (disposed)

```

```

    {
        throw new ObjectDisposedException(...);
    }

    // Now call some native methods using the resource
    ...
}

public void Dispose()
{
    Dispose(true);
    GC.SuppressFinalize(this);
}

protected virtual void Dispose(bool disposing)
{
    // Protect from being called multiple times.
    if (disposed)
    {
        return;
    }

    if (disposing)
    {
        // Clean up all managed resources.
        if (resource != null)
        {
            resource.Dispose();
        }
    }

    disposed = true;
}
}

' VB.NET sample:
Public Class DisposableResourceHolder
    Implements IDisposable

    Private disposed As Boolean = False
    Private resource As SafeHandle ' Handle to a resource

    Public Sub New()
        resource = ... ' Allocates the native resource
    End Sub

    Public Sub DoSomething()

```

```

        If (disposed) Then
            Throw New ObjectDisposedException(...)
        End If

        ' Now call some native methods using the resource
        ...
    End Sub

    Public Sub Dispose() Implements IDisposable.Dispose
        Dispose(True)
        GC.SuppressFinalize(Me)
    End Sub

    Protected Overridable Sub Dispose(ByVal disposing As Boolean)
        ' Protect from being called multiple times.
        If disposed Then
            Return
        End If

        If disposing Then
            ' Clean up all managed resources.
            If (resource IsNot Nothing) Then
                resource.Dispose()
            End If
        End If

        disposed = True
    End Sub

End Class

```

- ✅ **Do** implement the Basic Dispose Pattern on types containing instances of disposable types.
- ✅ **Do** extend the Basic Dispose Pattern to provide a finalizer on types holding resources that need to be freed explicitly and that do not have finalizers. For example, the pattern should be implemented on types storing unmanaged memory buffers.
- ✅ **You should** implement the Basic Dispose Pattern on classes that themselves don't hold unmanaged resources or disposable objects but are likely to have subtypes that do. A great example of this is the `System.IO.Stream` class. Although it is an abstract base class that doesn't hold any resources, most of its subclasses do and because of this, it implements this pattern.
- ✅ **Do** declare a protected virtual `void Dispose(bool disposing)` method to centralize all logic related to releasing unmanaged resources. All resource cleanup should occur in this method. The method is called from both the finalizer and the `IDisposable.Dispose` method. The parameter will be false if being invoked from inside a finalizer. It should be used to ensure any code running during finalization is not accessing other finalizable objects. Details of implementing finalizers are described in the next section.

```
// C# sample:
protected virtual void Dispose(bool disposing)
{
    // Protect from being called multiple times.
    if (disposed)
    {
        return;
    }

    if (disposing)
    {
        // Clean up all managed resources.
        if (resource != null)
        {
            resource.Dispose();
        }
    }

    disposed = true;
}

' VB.NET sample:
Protected Overridable Sub Dispose(ByVal disposing As Boolean)
    ' Protect from being called multiple times.
    If disposed Then
        Return
    End If

    If disposing Then
        ' Clean up all managed resources.
        If (resource IsNot Nothing) Then
            resource.Dispose()
        End If
    End If

    disposed = True
End Sub
```

✅ Do implement the IDisposable interface by simply calling Dispose(true) followed by GC.SuppressFinalize(this). The call to SuppressFinalize should only occur if Dispose(true) executes successfully.

```
// C# sample:
public void Dispose()
{
    Dispose(true);
    GC.SuppressFinalize(this);
}
```

```

}

' VB.NET sample:
Public Sub Dispose() Implements IDisposable.Dispose
    Dispose(True)
    GC.SuppressFinalize(Me)
End Sub

```

❌ **Do not** make the parameterless `Dispose` method virtual. The `Dispose(bool)` method is the one that should be overridden by subclasses.

❌ **You should not** throw an exception from within `Dispose(bool)` except under critical situations where the containing process has been corrupted (leaks, inconsistent shared state, etc.). Users expect that a call to `Dispose` would not raise an exception. For example, consider the manual try-finally in this C# snippet:

```

TextReader tr = new StreamReader(File.OpenRead("foo.txt"));
try
{
    // Do some stuff
}
finally
{
    tr.Dispose();
    // More stuff
}

```

If `Dispose` could raise an exception, further finally block cleanup logic will not execute. To work around this, the user would need to wrap every call to `Dispose` (within their finally block!) in a try block, which leads to very complex cleanup handlers. If executing a `Dispose(bool disposing)` method, never throw an exception if disposing is false. Doing so will terminate the process if executing inside a finalizer context.

✅ **Do** throw an `ObjectDisposedException` from any member that cannot be used after the object has been disposed.

```

// C# sample:
public class DisposableResourceHolder : IDisposable
{
    private bool disposed = false;
    private SafeHandle resource; // Handle to a resource

    public void DoSomething()
    {
        if (disposed)
        {
            throw new ObjectDisposedException(...);
        }
    }
}

```

```

        // Now call some native methods using the resource
        ...
    }

protected virtual void Dispose(bool disposing)
{
    if (disposed)
    {
        return;
    }

    // Cleanup
    ...

    disposed = true;
}
}

' VB.NET sample:
Public Class DisposableResourceHolder
    Implements IDisposable

    Private disposed As Boolean = False
    Private resource As SafeHandle ' Handle to a resource

    Public Sub DoSomething()
        If (disposed) Then
            Throw New ObjectDisposedException(...)
        End If

        ' Now call some native methods using the resource
        ...
    End Sub

    Protected Overridable Sub Dispose(ByVal disposing As Boolean)
        ' Protect from being called multiple times.
        If disposed Then
            Return
        End If

        ' Cleanup
        ...

        disposed = True
    End Sub

End Class

```

4.12.3 Finalizable Types

Finalizable types are types that extend the Basic Dispose Pattern by overriding the finalizer and providing finalization code path in the Dispose(bool) method. The following code shows an example of a finalizable type:

```
// C# sample:
public class ComplexResourceHolder : IDisposable
{
    bool disposed = false;
    private IntPtr buffer; // Unmanaged memory buffer
    private SafeHandle resource; // Disposable handle to a resource

    public ComplexResourceHolder()
    {
        this.buffer = ... // Allocates memory
        this.resource = ... // Allocates the resource
    }

    public void DoSomething()
    {
        if (disposed)
        {
            throw new ObjectDisposedException(...);
        }

        // Now call some native methods using the resource
        ...
    }

    ~ComplexResourceHolder()
    {
        Dispose(false);
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        // Protect from being called multiple times.
        if (disposed)
        {
            return;
        }
    }
}
```

```

        if (disposing)
        {
            // Clean up all managed resources.
            if (resource != null)
            {
                resource.Dispose();
            }
        }

        // Clean up all native resources.
        ReleaseBuffer(buffer);

        disposed = true;
    }
}

' VB.NET sample:
Public Class DisposableResourceHolder
    Implements IDisposable

    Private disposed As Boolean = False
    Private buffer As IntPtr ' Unmanaged memory buffer
    Private resource As SafeHandle ' Handle to a resource

    Public Sub New()
        buffer = ... ' Allocates memory
        resource = ... ' Allocates the native resource
    End Sub

    Public Sub DoSomething()
        If (disposed) Then
            Throw New ObjectDisposedException(...)
        End If

        ' Now call some native methods using the resource
        ...
    End Sub

    Protected Overrides Sub Finalize()
        Dispose(False)
        MyBase.Finalize()
    End Sub

    Public Sub Dispose() Implements IDisposable.Dispose
        Dispose(True)
        GC.SuppressFinalize(Me)
    End Sub

```



```

Protected Overridable Sub Dispose(ByVal disposing As Boolean)
    ' Protect from being called multiple times.
    If disposed Then
        Return
    End If

    If disposing Then
        ' Clean up all managed resources.
        If (resource IsNot Nothing) Then
            resource.Dispose()
        End If
    End If

    ' Clean up all native resources.
    ReleaseBuffer(Buffer)

    disposed = True
End Sub

End Class

```

☑ **Do** make a type finalizable, if the type is responsible for releasing an unmanaged resource that does not have its own finalizer. When implementing the finalizer, simply call `Dispose(false)` and place all resource cleanup logic inside the `Dispose(bool disposing)` method.

```

// C# sample:
public class ComplexResourceHolder : IDisposable
{
    ...
    ~ComplexResourceHolder()
    {
        Dispose(false);
    }

    protected virtual void Dispose(bool disposing)
    {
        ...
    }
}

' VB.NET sample:
Public Class DisposableResourceHolder
    Implements IDisposable

    ...
    Protected Overrides Sub Finalize()

```

```

        Dispose(False)
        MyBase.Finalize()
    End Sub

    Protected Overridable Sub Dispose(ByVal disposing As Boolean)
        ...
    End Sub

End Class

```

✅ **Do** be very careful to make type finalizable. Carefully consider any case in which you think a finalizer is needed. There is a real cost associated with instances with finalizers, from both a performance and code complexity standpoint.

✅ **Do** implement the Basic Dispose Pattern on every finalizable type. See the previous section for details on the basic pattern. This gives users of the type a means to explicitly perform deterministic cleanup of those same resources for which the finalizer is responsible.

✅ **You should** create and use a critical finalizable object (a type with a type hierarchy that contains `CriticalFinalizerObject`) for situations in which a finalizer absolutely must execute even in the face of forced application domain unloads and thread aborts.

✅ **Do** prefer resource wrappers based on `SafeHandle` or `SafeHandleZeroOrMinusOneIsInvalid` (for Win32 resource handle whose value of either 0 or -1 indicates an invalid handle) to writing finalizer by yourself to encapsulate unmanaged resources where possible, in which case a finalizer becomes unnecessary because the wrapper is responsible for its own resource cleanup. Safe handles implement the `IDisposable` interface, and inherit from `CriticalFinalizerObject` so the finalizer logic will absolutely execute even in the face of forced application domain unloads and thread aborts.

```

/// <summary>
/// Represents a wrapper class for a pipe handle.
/// </summary>
[SecurityCritical(SecurityCriticalScope.Everything),
HostProtection(SecurityAction.LinkDemand, MayLeakOnAbort = true),
SecurityPermission(SecurityAction.LinkDemand, UnmanagedCode = true)]
internal sealed class SafePipeHandle : SafeHandleZeroOrMinusOneIsInvalid
{
    private SafePipeHandle()
        : base(true)
    {
    }

    public SafePipeHandle(IntPtr preexistingHandle, bool ownsHandle)
        : base(ownsHandle)
    {
        base.SetHandle(preexistingHandle);
    }
}

```

```

[ReliabilityContract(Consistency.WillNotCorruptState, Cer.Success),
DllImport("kernel32.dll", CharSet = CharSet.Auto, SetLastError = true)]
[return: MarshalAs(UnmanagedType.Bool)]
private static extern bool CloseHandle(IntPtr handle);

protected override bool ReleaseHandle()
{
    return CloseHandle(base.handle);
}
}

/// <summary>
/// Represents a wrapper class for a local memory pointer.
/// </summary>
[SuppressUnmanagedCodeSecurity,
HostProtection(SecurityAction.LinkDemand, MayLeakOnAbort = true)]
internal sealed class SafeLocalMemHandle : SafeHandleZeroOrMinusOneIsInvalid
{
    public SafeLocalMemHandle()
        : base(true)
    {
    }

    public SafeLocalMemHandle(IntPtr preexistingHandle, bool ownsHandle)
        : base(ownsHandle)
    {
        base.SetHandle(preexistingHandle);
    }

    [ReliabilityContract(Consistency.WillNotCorruptState, Cer.Success),
DllImport("kernel32.dll", CharSet = CharSet.Auto, SetLastError = true)]
    private static extern IntPtr LocalFree(IntPtr hMem);


    protected override bool ReleaseHandle()
    {
        return (LocalFree(base.handle) == IntPtr.Zero);
    }
}

```

❌ Do not access any finalizable objects in the finalizer code path, as there is significant risk that they will have already been finalized. For example, a finalizable object A that has a reference to another finalizable object B cannot reliably use B in A's finalizer, or vice versa. Finalizers are called in a random order (short of a weak ordering guarantee for critical finalization).

It is OK to touch unboxed value type fields.

Also, be aware that objects stored in static variables will get collected at certain points during an application domain unload or while exiting the process. Accessing a static variable that refers to a finalizable object (or calling a static method that might use values stored in static variables) might not be safe if `Environment.HasShutdownStarted` returns true.

 **Do not** let exceptions escape from the finalizer logic, except for system-critical failures. If an exception is thrown from a finalizer, the CLR may shut down the entire process preventing other finalizers from executing and resources from being released in a controlled manner.

4.12.4 Overriding Dispose

If you're inheriting from a base class that implements `IDisposable`, you must implement `IDisposable` also. Always call your base class's `Dispose(bool)` so it cleans up.

```
public class DisposableBase : IDisposable
{
    ~DisposableBase()
    {
        Dispose(false);
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        // ...
    }
}

public class DisposableSubclass : DisposableBase
{
    protected override void Dispose(bool disposing)
    {
        try
        {
            if (disposing)
            {
                // Clean up managed resources.
            }

            // Clean up native resources.
        }
        finally
        {
            // ...
        }
    }
}
```

```

        base.Dispose(disposing);
    }
}

```

4.13 Interop

4.13.1 P/Invoke

✓ **Do** consult [P/Invoke Interop Assistant](#) and <http://pinvoke.net> to write P/Invoke signatures.

✓ **You can** use `IntPtr` for manual marshaling. By declaring parameters and fields as `IntPtr`, you can boost performance, albeit at the expense of ease of use, type safety, and maintainability. Sometimes it is faster to perform manual marshaling by using methods available on the `Marshal` class rather than to rely on default interop marshaling. For example, if large arrays of strings need to be passed across an interop boundary, but the managed code needs only a few of those elements, you can declare the array as `IntPtr` and manually access only those few elements that are required.

✗ **Do not** aggressively pin short-lived objects. Pinning short-lived objects unnecessarily extends the life of a memory buffer beyond the duration of the P/Invoke call. Pinning prevents the garbage collector from relocating the bytes of the object in the managed heap, or relocating the address of a managed delegate. However, it is acceptable to pin long-lived objects, which are ideally created during application initialization, because they are not moved relative to short-lived objects. It is costly to pin short-lived objects for a long period of time, because compacting occurs most in Generation 0 and the garbage collector cannot relocate pinned objects. This results in inefficient memory management that can adversely affect performance. For more information about copying and pinning, see <http://msdn.microsoft.com/en-us/library/23acw07k.aspx>.

✓ **Do** set `CharSet = CharSet.Auto` and `SetLastError = true` in the P/Invoke signature. For example,

```

// C# sample:
[DllImport("kernel32.dll", CharSet = CharSet.Auto, SetLastError = true)]
public static extern SafeFileMappingHandle OpenFileMapping(
    FileMapAccess dwDesiredAccess, bool bInheritHandle, string lpName);

' VB.NET sample:
<DllImport("kernel32.dll", CharSet:=CharSet.Auto, SetLastError:=True)> _
Public Shared Function OpenFileMapping( _
    ByVal dwDesiredAccess As FileMapAccess, _
    ByVal bInheritHandle As Boolean, _
    ByVal lpName As String) _
    As SafeFileMappingHandle
End Function

```

✓ **You should** wrap unmanaged resources in `SafeHandle` classes. The `SafeHandle` class is discussed in the [Finalizable Types](#) section. For example, the handle of file mapping is wrapped as follows.

```

/// <summary>

```

```

/// Represents a wrapper class for a file mapping handle.
/// </summary>
[SuppressUnmanagedCodeSecurity,
HostProtection(SecurityAction.LinkDemand, MayLeakOnAbort = true)]
internal sealed class SafeFileMappingHandle : SafeHandleZeroOrMinusOneIsInvalid
{
    [SecurityPermission(SecurityAction.LinkDemand, UnmanagedCode = true)]
    private SafeFileMappingHandle()
        : base(true)
    {
    }

    [SecurityPermission(SecurityAction.LinkDemand, UnmanagedCode = true)]
    public SafeFileMappingHandle(IntPtr handle, bool ownsHandle)
        : base(ownsHandle)
    {
        base.SetHandle(handle);
    }

    [ReliabilityContract(Consistency.WillNotCorruptState, Cer.Success),
DllImport("kernel32.dll", CharSet = CharSet.Auto, SetLastError = true)]
    [return: MarshalAs(UnmanagedType.Bool)]
    private static extern bool CloseHandle(IntPtr handle);

    protected override bool ReleaseHandle()
    {
        return CloseHandle(base.handle);
    }
}

''' <summary>
''' Represents a wrapper class for a file mapping handle.
''' </summary>
''' <remarks></remarks>
<SuppressUnmanagedCodeSecurity(), _
HostProtection(SecurityAction.LinkDemand, MayLeakOnAbort:=True)> _
Friend NotInheritable Class SafeFileMappingHandle
    Inherits SafeHandleZeroOrMinusOneIsInvalid

    <SecurityPermission(SecurityAction.LinkDemand, UnmanagedCode:=True)> _
    Private Sub New()
        MyBase.New(True)
    End Sub

    <SecurityPermission(SecurityAction.LinkDemand, UnmanagedCode:=True)> _
    Public Sub New(ByVal handle As IntPtr, ByVal ownsHandle As Boolean)
        MyBase.New(ownsHandle)
    End Sub

```

```

        MyBase.SetHandle(handle)
    End Sub

    <ReliabilityContract(Consistency.WillNotCorruptState, Cer.Success), _
    DllImport("kernel32.dll", CharSet:=CharSet.Auto, SetLastError:=True)> _
    Private Shared Function CloseHandle(ByVal handle As IntPtr) _
    As <MarshalAs(UnmanagedType.Bool)> Boolean
    End Function

    Protected Overrides Function ReleaseHandle() As Boolean
        Return SafeFileMappingHandle.CloseHandle(MyBase.handle)
    End Function

End Class

```

☑ **You should throw Win32Exception on the failure of P/Invoked functions that set the Win32 last error. If the function uses some unmanaged resources, free the resource in the finally block.**

```

// C# sample:
SafeFileMappingHandle hMapFile = null;
try
{
    // Try to open the named file mapping.
    hMapFile = NativeMethod.OpenFileMapping(
        FileMapAccess.FILE_MAP_READ,    // Read access
        false,                          // Do not inherit the name
        FULL_MAP_NAME                   // File mapping name
    );
    if (hMapFile.IsInvalid)
    {
        throw new Win32Exception();
    }

    ...
}
finally
{
    if (hMapFile != null)
    {
        // Close the file mapping object.
        hMapFile.Close();
        hMapFile = null;
    }
}

' VB.NET sample:
Dim hMapFile As SafeFileMappingHandle = Nothing

```


```


Try
    ' Try to open the named file mapping.
    hMapFile = NativeMethod.OpenFileMapping( _
        FileMapAccess.FILE_MAP_READ, _
        False, _
        FULL_MAP_NAME)
    If (hMapFile.IsInvalid) Then
        Throw New Win32Exception
    End If


    ...
Finally
    If (Not hMapFile Is Nothing) Then
        ' Close the file mapping object.
        hMapFile.Close()
        hMapFile = Nothing
    End If
End Try

```

4.13.2 COM Interop

 **Do not** force garbage collections with `GC.Collect` to release COM objects in performance sensitive APIs. A common approach for releasing COM objects is to set the RCW reference to null, and call `System.GC.Collect` followed by `System.GC.WaitForPendingFinalizers`. This is not recommended for performance reasons, because in many situations it can trigger the garbage collector to run too often. Code written by using this approach significantly compromises the performance and scalability of server applications. You should let the garbage collector determine the appropriate time to perform a collection.

 **You should** use `Marshal.FinalReleaseComObject` or `Marshal.ReleaseComObject` to manage the lifetime of an RCW manually. It has much better performance than forcing garbage collections with `GC.Collect`.

 **Do not** make cross-apartment calls. When you call a COM object from a managed application, make sure that the managed code's apartment matches the COM object's apartment type. By using matching apartments, you avoid the thread switch associated with cross-apartment calls.